DULLIN - RETZLAFF - SCHNEIDER
STRASSENBURG

CPC

CONSEJOS Y TRUCOS

UN POZO DE CIENCIA PARA
CPC 464, 664 Y 6128



UN LIBRO DATA BECKER
EDITADO POR FERRE MORET, S.A.

DULLIN - RETZLAFF - SCHNEIDER
STRASSENBURG

CPC

CONSEJOS Y TRUCOS

UN POZO DE CIENCIA PARA
CPC 464, 664 Y 6128



UN LIBRO DATA BECKER

EDITADO POR FERRE MORET, S.A.

Este libro ha sido traducido por Montserrat Sierra Urroz.

Imprime: APSSA, ROCA UMBERT, 26 - L'HOSPITALET DE LL. (Barcelona)

ISBN 84-86437-56-3

Depósito legal B-26.791/86

Advertencia importante

Los circuitos, procedimientos y programas reproducidos en este libro son divulgados sin tener en cuenta el estado de las patentes. Están destinados exclusivamente al uso amateur o docente, y no pueden ser utilizados para fines comerciales. Todos los circuitos, datos técnicos y programas de este libro, han sido elaborados o recopilados con el mayor cuidado por el autor y reproducidos utilizando medidas de control eficaces. No obstante, es posible que exista algún error. FERRE MORET, S.A. se ve por tanto obligada a advertirles, que no puede asumir ninguna garantía, ni responsabilidad jurídica, ni cualquier otra responsabilidad sobre las consecuencias atribuibles a datos erróneos. El autor les agradecerá en todo momento la comunicación de posibles fallos.

INDICE

INTRODUCCION

Estructura del libro

Este libro está dividido por sectores temáticos diferenciados en tres partes:

La primera parte se ocupa exclusivamente del BASIC del ordenador Amstrad. En ésta se presentan programas y rutinas BASIC útiles e interesantes, así como también algunos trucos para el manejo del BASIC, que sirven de ayuda para programas propios o para la aceleración de programas BASIC.
Al final de este apartado, encontrará una descripción exacta de instrucciones BASIC específicas para el Amstrad con ejemplos de su ámbito de aplicación.

La segunda parte contiene rutinas en lenguaje máquina, que pueden emplearse inmediatamente y que complementan las posibilidades del interpretador BASIC.
Aunque los programas presentados en este apartado están escritos en lenguaje máquina, tienen también su importancia para aquellos lectores que no estén especialmente interesados en el lenguaje máquina.
Aunque este apartado presenta principalmente programas confeccionados para ser tecleados, encontrará también en él interesantes informaciones de fondo.

La tercera parte de este libro se ocupa del lenguaje máquina del ordenador Amstrad. Contiene una lista muy completa de rutinas interesantes del sistema operativo y del interpretador BASIC. Además presenta algunos consejos y trucos para la programación eficaz en lenguaje máquina (optimización del tiempo), así como por ejemplo el empleo de determinados mecanismos del sistema operativo del ordenador Amstrad, y la ampliación de instrucciones mediante RSX.

El capítulo 23 de este apartado describe un método hasta ahora desconocido, según nuestras informaciones, para la reubicación (desplazamiento con adaptación de direcciones absolutas) de programas en lenguaje máquina. Este capítulo ha sido pensado especialmente para aclarar las técnicas allí utilizadas, en torno a la modificación de un programa por medio de un programa y un extendido truco que utiliza la pila (stack). Seguramente en este capítulo, los programadores en lenguaje máquina más adelantados, aprenderán algo más sobre la "programación en lenguaje máquina con estilo".
Aunque este tercer apartado se ocupe exclusiva y prácticamente del lenguaje máquina, puede ser también de interés para el lector, que hasta este momento no haya tenido ningún contacto con este tipo de lenguaje. En especial los dos primeros capítulos de esta parte sirven de introducción a este grupo de lectores, conteniendo explicaciones exhaustivas y programas de ejemplo claros y concisos gracias a su reducida extensión.

La notación de números hexadecimales

Los números hexadecimales se expresan en este libro precedidos del signo &. Esta es la notación que se utiliza también para el BASIC Amstrad; nosotros la hemos aceptado por motivos de estandarización. Algunos ensambladores (Assemblers) utilizan otras notaciones (por ejemplo, un "#" antes, o "H" después). En este caso se debe poner atención en la adaptación correspondiente al entrar el listado-fuente.

Signos especiales en los listados

En este libro se encontrará a menudo con el carácter "^". No dude a la hora de buscar este carácter en el teclado. Corresponde a la flecha hacia arriba, que en el BASIC se utiliza como operador de potenciación. Este carácter "^" lo edita la impresora a modo de flecha.

El teclado del CPC

Desgraciadamente algunos caracteres de ciertas teclas no se corresponden en los tres tipos de CPC. Siempre que se hable en este libro de la tecla RETURN (correspondiente al 6128), en el 664 y el 464 se trata de la tecla ENTER que se encuentra en el bloque central del teclado.

PARTE 1

Consejos & trucos para el BASIC

1. Métodos de ordenación

¿Por qué nos ocupamos de este tema tratándose de programas relativamente poco inteligibles y de dudosa aplicación?.

Es especialmente interesante el análisis de algoritmos de ordenación en relación a la velocidad de ejecución conseguida. Una parte importante del tiempo de cálculo utilizado por el ordenador se usa para la ordenación. Apenas existen programas de aplicaciones, por muy sencillos que sean, que no utilicen una función de ordenación. No en vano se buscan continuamente nuevos y mejores procedimientos de ordenación.

Este capítulo debería ser, en principio, breve. Sin embargo, al introducirnos más en el tema, resultó ser tan interesante, que decidimos dedicarle una mayor atención.

Especialmente los algoritmos de ordenación son a menudo procedimientos ingeniosos, cuya programación ocupa en realidad poco espacio. Justo en los programas más simples de menos de 10 líneas, es donde más sorprende la diferencia de rendimiento. La dificultad no se encuentra pues en la realización de programas en algún que otro lenguaje, sino mucho más en un procedimiento efectivo para el desarrollo de la ordenación de datos.

Existen dos factores que determinan en gran medida el tiempo de cálculo en la ordenación.

1. El número medio de comparaciones a efectuar
2. El número medio de desplazamientos o intercambios de datos.

Es conveniente que estos dos factores, especialmente la cantidad de comparaciones, sean lo más pequeños posible.

La calidad de un algoritmo se determina por estas magnitudes. El problema es que el tiempo de ordenación no es proporcional a la cantidad de datos a ordenar. Esto significa:

Si ordenar 10 nombres dura un segundo, no se dobla necesariamente el tiempo de ordenación en el caso de 20 nombres. El tiempo se eleva al cuadrado con algoritmos no adecuados. Con una cantidad doble de datos, se necesita un tiempo cuádruple ($4=2^2$), con una cantidad diez veces mayor, se utiliza un tiempo 100 veces mayor ($100=10^2$).

Vayamos a aclarar el procedimiento de ordenación más simple, el llamado "método de burbuja". La ordenación de burbuja debe su nombre a la particularidad de que los elementos mayores ascienden durante el proceso al igual que las burbujas en el agua. Además los mayores son los que tienen un mayor ascenso. De este modo se consigue finalmente un campo ordenado.

Con ello, los elementos se irán comparando entre sí de dos en dos. Si el elemento que se encuentra debajo es mayor que su anterior, se intercambiarán, así el elemento mayor sube hacia arriba como una burbuja. El mismo procedimiento se realizará con los dos elementos siguientes, y así sucesivamente. Este procedimiento finaliza cuando se ha tratado el campo completo. Después de este proceso, el elemento mayor se encuentra arriba del todo, en el lugar donde le corresponde.

Si nuestro campo consta de n elementos, en este proceso se llevarán a cabo n-1 comparaciones.

Como el elemento mayor se encuentra en su lugar, ya podemos prescindir de él. A continuación empezaremos de nuevo el proceso, considerando que el campo a ordenar sólo tiene n-1 elementos. Después del segundo proceso, el segundo elemento mayor también se encuentra en su sitio. Para ello hemos necesitado n-2, o sea, en total (n-1)+(n-2) comparaciones. El tamaño del campo a ordenar disminuye de nuevo en 1, y se vuelve empezar hasta llegar a la ordenación completa.

Para ello debemos realizar

(n-1)+(n-2)+(n-3)+...+2+1=n*(n-1)/2

comparaciones. Para n=10 se necesitan 45 comparaciones, para n=100 (la cantidad 10 veces mayor) 4950, o sea, más de 100 veces mayor. ¡El tiempo de ordenación se eleva al cuadrado!
En los siguientes programas se encuentran las siguientes indicaciones:

1. El último campo a ordenar se encuentra en a(anz).

2. El índice máximo de campo se encuentra almacenado en "anz". Con ello, el número de elementos a ordenar es anz+1, pues con el índice 0 también se encuentra almacenado un elemento.

3. Después del proceso de ordenación,
   el campo ordenado se encuentra de nuevo en a(anz).

4. Todas las variables que no estén señalizadas con "$" o "!", son variables enteras. Esto se hizo para ahorrar tiempo.

Veamos ahora el programa base, que realiza las preparaciones necesarias.

```
10 'Ordenaciones
20 DEFINT a-z:DEFREAL t
30 RANDOMIZE TIME
40 aus=0:'Flag para salida procedimiento ordenacion
50 auga=0:'Numero canal para salida
60 anz=20:anz=anz-1:'Magnitud de campo
70 DIM a(anz+1),b(anz),l(20),r(20)
80 FOR i=0 TO anz:a(i)=INT(100*RND):b(i)=a(i):NEXT:'Crear ca
mpo a ordenar
90 READ j$:IF j$="#" THEN END ELSE j=VAL(J$):PRINT J,:IF aus
 THEN PRINT:'Leer numeros de programa
100 FOR i=0 TO anz:a(i)=b(i):NEXT:t=TIME
110 ON j GOSUB 200,300,400,500,600,700,800,900,1050,1200
120 PRINT#auga,(TIME-t)/300
130 GOTO 90:' Signo de final
140 DATA 1,2,3,4,5,6,7,8,9,10:'Programas a ejecutar
150 DATA #
160 REM SUB Para salida procedimiento ordenacion
170 FOR n=0 TO anz:PRINT #auga,USING"##";a(n);:PRINT #auga,"
  ";:NEXT
180 PRINT#auga
190 RETURN
```

Ahora vayamos a la ordenación de burbuja:

```
200 REM Ordenacion de burbuja
210 FOR i=anz TO 1 STEP -1
220 FOR j=1 TO i
230 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s
240 IF aus THEN GOSUB 170
250 NEXT j,i
260 RETURN
```

En cada recorrido, el bucle exterior disminuye en uno el índice del último campo a ordenar. En el bucle interior se comparan unos con otros, y por pares, todos los elementos restantes del campo , cambiando eventualmente su posición.

Para obtener valores temporales correctos, debe suprimir la línea 240 del programa. Esta sirve para la salida de los campos después de cada comparación. Esta salida se produce cuando la variable se pone a -1. Así podrá seguir el desarrollo durante la ordenación.

Ocúpese ahora de las siguientes ampliaciones:

Construya un contador para el número de comparaciones y otro para el número de cambios de posición. Analice los valores obtenidos en relación con el tiempo y los valores teóricos. Calcule Ud. la porción de tiempo para cambios de posición en relación con la totalidad del tiempo utilizado (ver también el capítulo: Aceleración de programas BASIC).

¡La ordenación de burbuja es un mal procedimiento de ordenación!

Pero precisamente este ejemplo nos demuestra, que modificaciones insignificantes pueden mejorar un algoritmo de modo significativo.

La ordenación de burbuja tiene el inconveniente de que sigue ordenando aunque todo se encuentre ya en su posición correcta. Esto se puede comprobar fácilmente. Si se efectúa un cambio de posición, activamos un flag (bandera). Si al final del proceso, el flag no está aún activado, o sea, no se ha efectuado ningún intercambio, el trabajo está listo.

Analice, basándose en la medición del tiempo, las comparaciones e intercambios, y así la media de mejoramiento que esto comporta.

Esta idea del flag puede ampliarse más. En lugar de activar simplemente un flag, almacenemos el índice del intercambio. Después de una pasada, el flag contiene el índice del último intercambio. Todos los elementos con índice más alto se encuentran ya correctamente ordenados. Así pues, ahora sólo debemos recorrer el campo hasta este índice.

```
300 REM Ordenacion de burbuja ampliada
310 FOR i=anz TO 1 STEP -1
320 maxj=0
330 FOR j=1 TO i
340 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s:maxj=j
350 IF aus THEN GOSUB 170
360 NEXT j
370 IF maxj=0 THEN RETURN ELSE i=maxj
380 NEXT i
390 RETURN
```

También aquí se vuelven a poner de manifiesto las mejoras que se producen con los métodos mencionados respecto al procedimiento anterior.

Intente escribir aún otra "versión de burbuja", donde intercambiando, "se ordene hacia arriba", esto es, el elemento mayor va hacia arriba, y "se ordene hacia abajo", el elemento menor va hacia abajo.

Utilice dos indicadores que limiten el campo que todavía no se ha ordenado. Analice este procedimiento basándose en su rendimiento.

Con esto es suficiente para la ordenación de burbuja. Sólo quedan algunos algoritmos simples, que se diferencian por su tratamiento. Lleve a cabo los tests de comparación de cada uno de los algoritmos.

Para poder comprender mejor el procedimiento desde el principio, se encuentra en cada programa la opción de salida. Para llevar a cabo los análisis de velocidad, debería borrar esta línea.

Ordenación Max

La ordenación Max recorre cada campo sin ordenar en busca de su mayor elemento. Si encuentra un elemento, que es mayor, o sea que se encuentra almacenado al final del campo, se efectúa un intercambio entre ambos. Después de cada proceso, el campo restante se disminuye en 1.

```
400 REM Ordenacion Max.
410 FOR i=anz TO 1 STEP -1
420 FOR j=0 TO i-1
430 IF a(j)>a(i) THEN s=a(j):a(j)=a(i):a(i)=s
440 IF aus THEN GOSUB 170
450 NEXT j,i
460 RETURN
```

Ordenación straight u ordenación por elección

Este procedimiento es básicamente un procedimiento de ordenación Max mejorado. La mejora consiste en que cuando se encuentra un elemento mayor, no se efectúa un intercambio automático.

En lugar de ello, se busca el máximo en el campo restante y luego se realiza el cambio de posición. El número de comparaciones es igual de pequeño que el de intercambios.

```
500 REM Straight
510 FOR i=anz TO 1 STEP -1:m=0
520 FOR j=0 TO i
530 IF a(j)>m THEN m=a(j):k=j
540 NEXT
550 a=a(i):a(i)=a(k):a(k)=a
560 IF aus THEN GOSUB 170
570 NEXT
580 RETURN
```

Ordenación por inserción (Insertsort)

La ordenación por inserción es un método , que todos conocemos. En el juego de cartas, ordenamos cada nueva carta, y la introducimos entre las que ya teníamos. Del mismo modo funciona el procedimiento de ordenación por inserción.
La nueva carta se tomará provisionalmente como si fuera la mayor, y se comparará con sus vecinas, si es menor, se cambiará su posición. Por consiguiente, se empezará con dos elementos a construir un campo ordenado de tamaño cada vez mayor.

```
600 REM Ordenacion por insercion
610 FOR i=1 TO anz
620 FOR j=i TO 1 STEP -1
630 IF a(j)>=a(j-1) THEN 670
640 s=a(j-1):a(j-1)=a(j):a(j)=s
650 IF aus THEN GOSUB 170
660 NEXT j
670 NEXT i
680 RETURN
```

Como un desplazamiento es más rápido que un intercambio, se puede mejorar la ordenación por inserción del mismo modo que la ordenación Max:
Primero se averigua la posición que pertenece al nuevo elemento a ordenar. Después se desplazarán todos los elementos mayores y finalmente se situará el elemento en el lugar correspondiente.

```
700 REM Ordenacion por insercion mejorada
710 FOR i=1 TO anz
720 a=a(i)
730 FOR j=i-1 TO 0 STEP -1
740 IF a<a(j) THEN NEXT j
750 FOR k=i TO j+2 STEP -1:a(k)=a(k-1):NEXT k
760 a(j+1)=a
770 IF aus THEN GOSUB 170
780 NEXT i
790 RETURN
```

Esta versión se puede cambiar aún. El desplazamiento del campo puede efectuarse más rápidamente que en la versión anterior, en campos más pequeños y con un bucle extra.

```
800 REM 3a Version ordenacion por insercion
810 FOR i=1 TO anz
820 a=a(i):j=i-1
830 IF a>=a(j) THEN 860
840 a(j+1)=a(j):j=j-1
850 IF j>=0 THEN 830
860 a(j+1)=a
870 IF aus THEN GOSUB 170
880 NEXT
890 RETURN
```

Binary Sort = Inserción con úsqueda binaria

En el último programa se gastaba una gran parte del tiempo para encontrar la posición de un nuevo elemento en una lista para ordenar. El gasto de tiempo para esta búsqueda se puede reducir considerablemente con la ayuda de la búsqueda binaria. En la sencilla ordenación por inserción la comparación se realiza paso a paso con cada elemento. La ordenación binaria en cambio, es un procedimiento de búsqueda "inteligente".
Imagínese que debe adivinar a través de preguntas, un número entre 1 y 128. Según el "método de ordenación insert", en este ejemplo se debería preguntar:

¿Es el número 128?
¡no!
¿Es el número 127?
¡no!
¿Es el número 126?
.
.
.

Pero sería más efectivo preguntar:

¿Es el número mayor, menor o igual que 64?
¡Mayor que 64!

La siguiente pregunta sería:

¿Es el número mayor, menor o igual que 96?
¡Menor que 96!

Luego:

¿Es el número mayor, menor o igual que 84?
¡Menor que 84!

En estas consultas se sigue el principio de digamos "cercar", o sea "encerrar" el número buscado. Mediante preguntas sistemáticas, se limita el número de posibles respuestas.

Un método similar de preguntas podemos observarlo por ejemplo en juegos de adivinanza.

Pregunta:"¿Es de sexo masculino o femenino el candidato?"

Posibles respuestas en caso normal: masculino o femenino.

Aquí existen pues dos tipos de respuesta. Lo más importante es que el ámbito de posibilidades para las preguntas se reduce muchísimo gracias a la respuesta.

El intervalo, donde el número se puede encontrar, se reduce cada vez a la mitad. El número que debemos encontrar es 30. Los intervalos después de cada pregunta son pues:

Pregunta	Respuesta	Intervalo
>, < o =64	<64	1-63
>, < o =32	<32	1-31
>, < o =16	>16	17-31
>, < o =24	>24	25-31
>, < o =28	>28	
>, < o =30	=30	¡¡encontrado!!!

Fueron necesarias 6 preguntas (comparaciones) para encontrar el número. Con el método anterior eran necesarias 98. El número máximo de comparaciones necesarias en n elementos con búsqueda binaria es proporcional a ln_2 n (ln_2=logaritmo en base 2).

El nombre del procedimiento proviene de que el intervalo que se cuestiona se reduce cada vez a la mitad. En base a esto, el logaritmo se tomará también en base 2. El ahorro es enorme en listas grandes.
La posición de un elemento que se encuentra en un campo a ordenar con 60 millones de elementos (número de habitantes de la RFA), se puede averiguar con 26 comparaciones. Observe exactamente el programa de ordenación binaria,

y en caso de duda, realice una simulación a mano (juegue a ser ordenador), para clarificar la forma de trabajo del algoritmo.

A causa de la estructura un poco complicada del programa, la ventaja en la velocidad es visiblemente superior a la de la ordenación por inserción para campos de un tamaño aproximado de 100 ó más elementos.

```
900 REM Ordenacion Binary
910 FOR i=1 TO anz
920 IF a(i)>a(i-1) THEN 1000
930 l=-1:r=i+1
940 h=INT((l+r)/2)
950 IF a(i)>a(h) THEN l=h ELSE r=h
960 IF r>l+1 THEN 940
970 a=a(i):FOR j=i TO r+1 STEP -1:a(j)=a(j-1):NEXT
980 a(r)=a
990 IF aus THEN GOSUB 170
1000 NEXT
1010 RETURN
```

Ordenación Shell

El algoritmo de la clasificación Shell lleva el nombre de su inventor. D.L.Shell, que desarrolló el algoritmo en los años 50. Este procedimiento trata de forma interesante la ordenación de método burbuja, con la simple pero importante diferencia de que el campo a ordenar, se divide en pequeños subcampos, que se ordenan uno por uno, y al final completan la ordenación del campo entero.

La idea de este procedimiento, es que se ordene de un modo aproximado. Tomemos por ejemplo un campo con 16 elementos. Para esta ordenación aproximada, se compararán el 1 y el 9, el 2 y el 10, el 3 y el 11 ...elementos, y en caso posible se efectuará un intercambio. Así se consigue la primera ordenación aproximada.

En este proceso se han comparado elementos con distancia 8.

En el proceso siguiente la distancia se reducirá a la mitad. Ahora se ordenarán los subcampos que resultan de los elementos con una distancia de 4, o sea, el subcampo del 1, 5, 9, y 13 elemento, el del 2, 6, 10, 14 elemento ... Ahora se ordenan los nuevos subcampos, donde se utiliza el procedimiento de ordenación por inserción. Así se comparará el 1 con el 5, y se efectuará un intercambio eventual, luego se ordenará el 9 en el 1 y el 5, y después el 13 entre el 1, 5, y 9. El mismo procedimiento se realizará con los otros subcampos. Ahora se reducirá de nuevo a la mitad la distancia de comparación, hasta llegar a uno. El último proceso de ordenación se llevará a cabo con el campo completo.

El procedimiento de ordenación Shell es un algoritmo muy interesante, el que ha llegado más lejos de los que hemos nombrado hasta ahora, si se deben ordenar campos con más de 20 elementos.

```
1050 REM Ordenacion Shell
1060 FOR m=anz-1 TO 1 STEP -1
1070 m=INT((m+1)/2)
1080 FOR j=0 TO anz-m
1090 i=j
1100 IF a(i)<=a(i+m) THEN 1150
1110 s=a(i):a(i)=a(i+m):a(i+m)=s
1120 IF aus THEN GOSUB 170
1130 i=i-m
1140 IF i>=0 THEN 1100 ELSE 1150
1150 NEXT j,m
1160 RETURN
```

Ordenación Quick

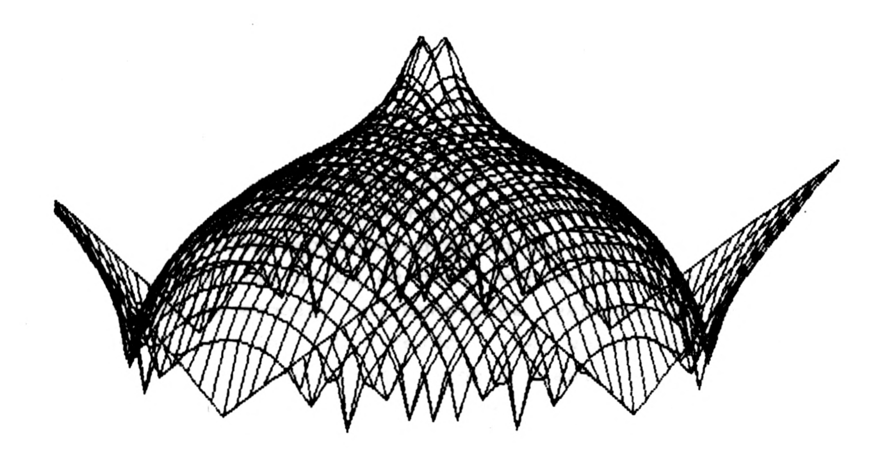
Uno de los mejores procedimientos de ordenación conocidos es el Quick(rápido). Los procedimientos presentados hasta ahora estaban pensados para campos con más de 50 elementos. La idea para el procedimiento de ordenación Quick es la siguiente:

El campo a ordenar se divide en dos partes. El elemento que queda entre las dos partes sirve como elemento de comparación. Se intercambiarán todos los elementos de la parte inferior, que sean mayores que el elemento de comparación. En algunos casos también se intercambiará el elemento de comparación. Esta ordenación aproximada es similar a una ampliación del procedimiento Shell.

Después del primer proceso, se empezará de nuevo el procedimiento con cada una de las dos partes. Esto es, el procedimiento es recursivo, es decir, se llama a sí mismo. Para posibilitar esta estructura recursiva también en BASIC, las variables más importantes se almacenarán antes de empezar de nuevo, sinó podrían perderse. Para ello, se encuentran los campos l(sp) y r(sp).

```
1200 REM Ordenacion rapida (Quicksort)
1210 sp=0:l  (sp)=0:r(sp)=anz
1220 l=l(sp):r=r(sp):sp=sp-1
1230 i=l:j=r:vv=a(INT((l+r)/2))
1240 WHILE a(i)<vv :i=i+1:WEND
1250 WHILE a(j)>vv :j=j-1:WEND
1260 IF i>j THEN 1300
1270 s=a(i): a(i)=a(j):a(j)=s:IF aus THEN GOSUB 170
1280 i=i+1:j=j-1
1290 IF i<=j THEN 1240
1300 IF i<r THEN sp=sp+1:l(sp)=i:r(sp)=r
1310 r=j:IF l<r THEN 1230
1320 IF sp>=0 THEN 1220
1330 RETURN
```

2. Gráficos 3 D



```
40 DEF FNf(x)=SQR(ABS((16-x*x-z*z)))+1/SQR(x*x+z*z+0.1)
50 ap=34
80 DATA -4,4,-2,6,-4,4
```

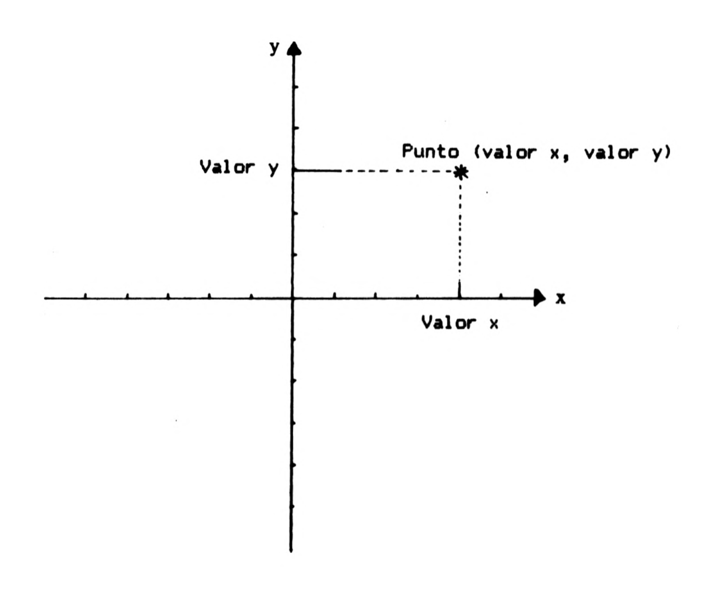
Una de las aplicaciones más interesantes y bellas del ordenador, es la producción de dibujos tridimensionales. La teoría completa de los gráficos 3-D es muy compleja y ha sido impulsada notablemente en los últimos años con el desarrollo de los sistemas CAD (Computer Aided Design). Con ayuda de las últimas generaciones de ordenadores, ya es posible la producción completa de películas, especialmente escenas de ciencia ficción, de mano de los ordenadores.

Pero aquí naturalmente, no nos podemos mover en estas dimensiones. El ordenador Amstrad ofrece grandes posibilidades dentro de su clase. La resolución de 640*200 puntos conseguida en MODE 2 presenta condiciones ideales para la representación de funciones tridimensionales.

Ocupémonos ahora un poco de la teoría, es decir, del concepto de función.

Una función es una representación gráfica. En ella, al menos en la bidimensional, a cada valor de un conjunto de valores (el campo de existencia o definición) le corresponde uno -y sólo uno- de un segundo conjunto de valores (el campo de variabilidad), denominado valor de función. La relación entre estas dos cantidades se denomina función.

Se puede interpretar una función como una tabla de valores, donde cada valor del campo de definición tiene asignado un valor de la función. Los pares de números obtenidos pueden representarse gráficamente como un punto del plano en un sistema de coordenadas, señalando los valores en dos rectas perpendiculares entre sí, obteniendo el punto por la intersección de las líneas horizontal y vertical perpendiculares a los ejes para cada valor. Cada par de valores representa pues las coordenadas de un punto. Veamos la siguiente función como ejemplo:



Para cada número decimal se representará su valor entero, despreciando los números que se encuentren detrás de la coma.

Algunos de los pares de valores de esta función son:

X	Y
1,5	1
3,7593	3
1000,379	1000
2	2
-1,5	-2

Queremos representar esta función mediante el ordenador.

Primero se nos presenta la tarea de transformar las instrucciones de dibujo en una forma "comprensible", esto es, matemática para el ordenador. Para nuestro ejemplo es simple. La función anterior se representará mediante la función INT del ordenador (parte entera).

La variable para el valor del campo de definición es X. La variable para el campo de valores es Y. La función debe representarse en el campo de -5 a 15. Un primer intento podría ser:

```
10 MODE 2
20 FOR X=-5 TO 15
30 Y=INT(X)
40 PLOT X,Y
50 NEXT
```

No conseguimos el efecto deseado. Debemos adaptar nuestro programa al formato de la pantalla del ordenador. La anchura de la pantalla (=dirección X) es de 640 puntos. La altura (=dirección Y) es de 200 puntos, que se denominan con las coordenadas de 0 a 400 y se reducen internamente primero a la mitad.

La imagen conseguida es una representación en miniatura de la imagen deseada. Aún debemos ampliarla en las direcciones X e Y correspondientes.

Se introducirán dos factores de escala que amplíen o reduzcan los valores de x e y, ofreciendo la posibilidad de rellenar la pantalla.

La función debe representarse en el intervalo de XMAX=15 a XMIN=-5. El campo a representar tendrá en total la magnitud de XMAX-XMIN=15-(-5)=20. Como disponemos de 640 puntos, 20 debe extenderse a 640, o sea, los valores X deben multiplicarse por 640/20=32.

```
10 MODE 2
20 XMIN=-5:XMAX=15
30 ESCX=640/(XMAX-XMIN)
40 FOR X=XMIN TO XMAX STEP 1/ESCX
50 Y=INT(X)
60 PLOT X*ESCX,Y
70 NEXT X
```

La instrucción STEP provoca que se dibuje también un punto para cada uno de los 640 valores X representables.

La imagen es un poco "plana", porque no hemos ampliado la escala del eje Y.

Igual que con el eje X, debemos saber entre qué límites superior e inferior, se debe dibujar en la recta del eje Y. Esto, en muchos casos,no es posible saberlo previamente, especialmente si se trata de una función desconocida. La solución es hacer primero una prueba, o realizar un test para encontrar los valores máximo y mínimo de Y. Luego se toman estos valores como límite.

En base a las particularidades de nuestra función, es fácil reconocer que Y puede moverse entre -5 y 15.

Añada las siguientes líneas, cambiando las antiguas.

```
21 YMIN=-5:YMAX=15
31 ESCY=400/(YMAX-YMIN)
```

Cambie la línea 60 por PLOT X*ESCX,Y*ESCY

A pesar de la introducción de los factores de escala (ESC), todavía no hemos conseguido que la función "utilice" todo el espacio de pantalla. Es necesario que traslademos el punto origen(0,0) de pantalla en función de los límites superiores e inferiores de X y Y. Normalmente este punto se encuentra en la esquina inferior izquierda de la pantalla. En principio podríamos trasladar el punto origen (inglés: Origin) con la instrucción ORIGIN indicada para ello. En nuestro caso sería posible con la línea

```
35 ORIGIN 160,100
```

Este procedimiento, que naturalmente es el más fácil, funciona sólo con la condición de que el origen se encuentre bastante cerca del punto de origen real. En cambio, si se tratase de representar p.e. sólo la parte de la función comprendida entre 2000 y 2040, este método falla, pues en el caso de un ORIGIN -32000, -20000 se produce un overflow. Los cálculos necesarios se introducirán pues mediante el programa. Es necesario "trasladar" el valor de XMIN en la dirección X y el de YMIN en la dirección Y. Los siguientes cambios en la línea 60 son suficientes (borre de nuevo la línea 35 e introduzca de nuevo ORIGIN 0,0):

```
60 PLOT (X-XMIN)*ESCX,(Y-YMIN)*ESCY
```

Introduzcamos ahora las siguientes líneas, así tendremos además los ejes de coordenadas en la pantalla, en caso de que se encuentren dentro de ella.

```
35 IF XMIN*XMAX>0 THEN 37
36 MOVE (X-XMIN)*ESCX,0:DRAW (X-XMIN)*ESCX,400
37 IF YMIN*YMAX>0 THEN 40
38 MOVE 0,(Y-YMIN)*ESCY:DRAW 640,(Y-YMIN)*ESCY
```

La función anterior INT(X) tiene una particularidad poco agradable, que en matemáticas se llama "Discontinuidad". Esto significa, dicho simplemente, que existen unos saltos durante la ejecución del gráfico, o dicho de otro modo, no se ejecuta de forma lisa. Ahora trataremos funciones continuas, pues contienen particularidades estupendas para la programación. Una de las funciones más simples es la que representa en sí misma el valor X. Como se puede observar, esta función presenta como gráfica una recta, que constituye una diagonal entre los ejes. Para ilustrar una función de variable hemos elegido una función cuadrática, con la forma simple de $y=x^2$. Primero definiremos la función con la ayuda de la instrucción DEF FN:

```
15 DEF FNY(X)=X^2
```

Modificación:

```
50 Y=FNY(X)
```

Puesto que el trazado de una función continua siempre transcurre con bastante "tranquilidad", es decir, con cambios lentos en un intervalo suficientemente corto del eje X, es posible renunciar a dibujar cada uno de los puntos en detalle. Así pues se pueden dibujar puntos a intervalos y unirlos mediante una recta. De este modo se obtiene una ganancia de tiempo de 10 a 50 veces, según la distancia entre los puntos.
Precisamente por esto, el programa de gráfico de red 3-D, que se encuentra al final de este capítulo, es superior a otros, que calculan todos los puntos. Por ello debemos efectuar los siguientes cambios en el programa:

```
16 DIPU=10:REM Distancia (X) entre puntos
```

Cambiar el número de línea 40, por 42, y modifique ambas.

```
40 MOVE 0,(FNY(XMIN)-YMIN)*ESCY
42 FOR X=XMIN TO XMAX STEP 1/ESCX*DIPU
```

Cambiar línea 60 por:

```
DRAW (X-XMIN)*ESCX,(Y-YMIN)*ESCY
```

Además deben cambiarse los límites de Y a causa de la nueva función por -10 y 250 (u otras).

Pruebe otras funciones cuadráticas:

Función	XMIN	XMAX	YMIN	YMAX
x^2-50	-10	15	-150	200
x^2+20	-5	10	-10	90
$(X-5)^2$	-5	15	-10	100
$(X+5)^2$	-10	10	-10	250
$(X-5)^2-50$	-5	15	-60	60

Como hemos visto, la representación de funciones bidimensionales es relativamente fácil. La representación gráfica de una función bidimensional se encuentra en un plano y por ello es fácil trasladarla al plano de la pantalla. En un gráfico tridimensional este traslado directo no es posible, pues en el plano de la pantalla sólo disponemos de dos dimensiones.

En una representación tridimensional se asignará a cada par de valores del campo de definición (ahora un plano) un valor del campo de variabilidad. Podríamos imaginar que el plano de definición es una mesa, en la que cada punto tiene asignada una determinada altura.

Para construir una imagen tridimensional real de la función, deberíamos disponer de un material moldeable, como p.e. plastelina, con el que se pudiera hacer una representación a partir de cada punto.

La tarea del programador consiste en representar esta figura tridimensional en un plano, o sea, en dos dimensiones. Una representación aproximada a la realidad debería tener en cuenta que las líneas más alejadas se observan más pequeñas, y a la inversa, las que se encuentran más cerca del usuario, mayores. Mediante este efecto se puede conseguir la representación de una perspectiva real.

Para nuestro propósito, la representación de funciones, podemos pasar por alto este efecto. Imagínese que toma ese modelo de plastelina antes citado, y con un cuchillo grabamos una serie de líneas equidistantes, paralelas al borde delantero de la mesa, en su superficie. La tarea es dibujar estas mismas líneas (cortes) con el ordenador. La primera línea que se encuentra paralela a la arista de la mesa, se ha dibujado del mismo modo que hicimos con el anterior programa.

Consideremos esta función:

```
DEF FNY(X)=X^2+Z^2
```

donde Z corresponde al valor del tercer eje. En nuestro modelo corresponde a la arista de la mesa.

Supongamos, que la segunda línea grabada en la superficie atraviesa el valor -1 del eje Z o, dicho de otro modo, que se separa de la primera línea con una distancia de 1, mirando hacia atrás. Esta línea será trazada cuando Z se ponga a -1 y se ejecute el bucle normal para el trazado. Después Z tomará el valor de la siguiente línea de intersección, y así sucesivamente.

Primero cambie el número de la línea 40 por 41, y modifique la línea 15 con la definición de función anterior.

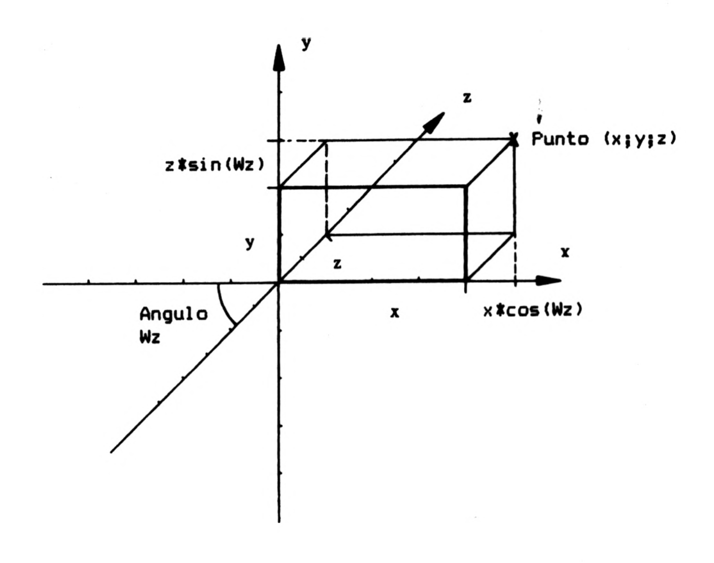
```
22 ZMIN=-10:ZMAX=0
40 FOR Z=ZMIN TO ZMAX STEP-1
```

Cambie la línea 41 por:

```
41 MOVE 0,(FNY(XMIN)-YMIN)*ESCY-Z*ESCZ*ESCY
70 NEXT
```

La imagen corresponde a nuestro modelo, si lo miramos frontalmente desde arriba =plano de la mesa. Pero nosotros deseamos poder elegir el ángulo de observación.

En la perspectiva estandar contemplamos nuestro modelo desde arriba a la derecha. En este caso, el eje Z transcurre en diagonal hacia atrás a la derecha. En la representación bidimensional, un punto que se encuentre sobre el extremo del eje Z, estaría situado en la esquina superior derecha de la pantalla. Esto significa, que los puntos, que en la realidad se encuentran más alejados, en una representación bidimensional aparecen trasladados en la pantalla hacia la derecha o hacia arriba sobre el eje de coordenadas X o Y de la pantalla. Cuanto más lejos del origen se encuentre el punto del eje Z, mayor será el desplazamiento sobre los ejes de coordenadas de la pantalla.



En el programa tenemos en cuenta:

Primero debemos cambiar la línea 41 por

```
41 MOVE -Z*ESCZ*ESCX,(FNY(XMIN)-YMIN)*ESCY-Z*ESCZ*ESCY
```

Y añadir:

```
6 DEG
25 ANG=45:'Angulo del eje Z con eje X
26 ESCX=COS(ANG):MAZY=SIN(ANG)
32 ESCZ=200/(ZMAX-ZMIN)
60 DRAW (X-XMIN)*ESCX-Z*ESCZ*ESCX,(Y-YMIN)*ESCY-Z*ESCZ*ESCY
17 DILI=10
```

Cambio:

```
40 FOR Z=ZMAX TO ZMIN STEP -1/ESCZ*DILI
```

Cambiemos todavía otras líneas para una nueva función y numeremos el programa de nuevo, obtendremos:

```
10 DEG
20 MODE 2
30 DEF FNy(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
40 dipu=20: REM distancia de punto
50 dili=20
60 xmin=-2:xmax=2
70 ymin=-7:ymax=7
80 zmin=-300:zmax=300
90 ang=45:'angulo del eje Z al eje X
100 eszx=COS(ang):eszy=SIN(ang)
110 escx=640/(xmax-xmin)
120 escy=400/(ymax-ymin)
130 escz=400/(zmax-zmin)
140 IF xmin*xmax>0 THEN 160
150 MOVE (-xmin)*escx,0:DRAW(-xmin)*escx,400
160 IF ymin*ymax>0 THEN 180
170 MOVE 0,(y-ymin)*escy:DRAW 640,(y-ymin)*escy
180 FOR z=zmax TO zmin STEP-1/escz*dili
190 MOVE -z*escz*eszx,(FNy(xmin)-ymin)*escy-z*escz*escy
200 FOR x=xmin TO xmax STEP 1/escx*dipu
210 y=FNy(x)
220 DRAW (x-xmin)*escx-z*escz*eszx,(y-ymin)*escy-z*escz*eszy
230 NEXT x,z
```

Este programa crea gráficos lineales tridimensionales. A menudo este método no ofrece resultados completos. Imagínese de nuevo nuestro modelo de plastelina, y realice de nuevo una serie de líneas paralelas a la arista de la mesa (eje Z). De este modo la superficie de la función se transformará en una red de líneas.

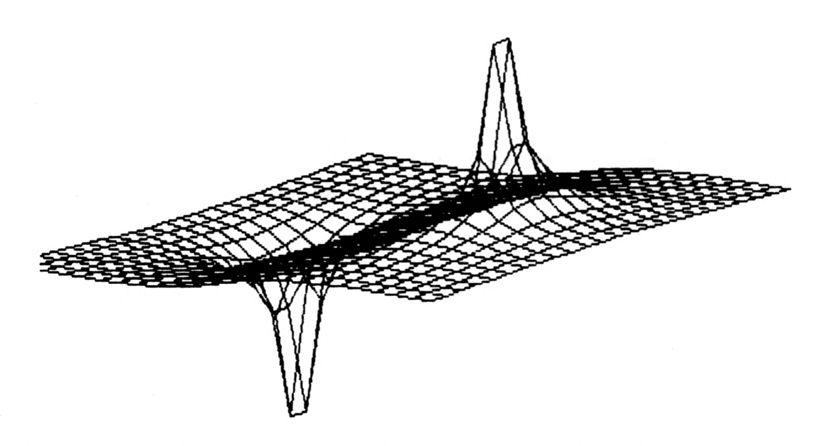
Para representar esto en dos dimensiones, podríamos simplemente añadir este programa al anterior programa en forma similar, con la diferencia, de que el bucle Z se intercambia con el bucle X. (Pruébelo)

Sin embargo, en este caso se calcularía cada punto de la red dos veces, hecho que naturalmente requiere más tiempo. El siguiente programa realiza las dos operaciones al mismo tiempo. Para ello se almacenan los últimos puntos trazados de la red, que serán unidos con los puntos correspondientes al trazar la línea siguiente. A continuación, se almacenarán los puntos que acaban de ser calculados, sustituyendo a los anteriores. De esta forma se traza la red completa. En el siguiente apartado presentamos el programa y algunos de los dibujos realizados con él.

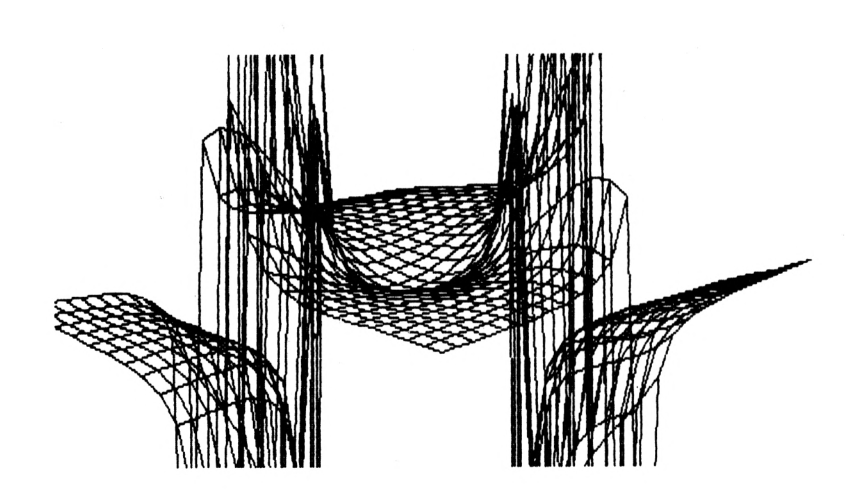
```
10 MEMORY &9FFF
20 MODE 2
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=15
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-7,7,-300,300
90 mx=400/(xe-xa)
100 my=400/(ye-ya)
110 mz=400/(ze-za)
120 DEG
130 al=42-22/2
140 as=22
150 zx=COS(al)^2
160 xx=COS(as)^2
170 zy=SIN(al)^2
180 xy=SIN(as)^2
190 j=0
200 FOR z=ze TO za STEP -(ze-za)/ap
210 i=0
220 FOR x=xe TO xa STEP -(xe-xa)/ap
230 y=FNf(x)
240 xk=120+xx*mx*(x-xa)-zx*mz*(z)
250 yk=my*(y-ya)-zy*mz*(z)-xy*mx*(x-xa)
260 IF i=0 THEN 280
270 MOVE xk,yk:DRAW x(i),y(i)
280 i=i+1
290 IF j=0 THEN 310
300 MOVE xk;yk:DRAW x(i),y(i)
310 x(i)=xk
320 y(i)=yk
330 NEXT x
340 j=j+1
350 NEXT z
360 IF INKEY$="" THEN 360
```

Lista de variables:

As:	Angulo entre la horizontal y el eje X
Al:	Angulo entre la horizontal y el eje Z
AP:	Número de puntos en la línea de red
I:	Contador
J:	Contador
MZ:	Escala Z
MY:	Escala Y
MX:	Escala X
XK:	Coordenadas X de pantalla
XY:	Factor de proyección X en el eje Y
XX:	Factor de proyección X en el eje X
XE:	Final de X
XA:	Inicio de X
X:	Valor de X
XK:	Coordenadas de Y
YE:	Final de Y
YA:	Inicio de Y
Y:	Valor de función de Y
ZY:	Factor de proyección Z en el eje Y
ZX:	Factor de proyección Z en el eje X
ZE:	Final de Z
ZA:	Comienzo de Z
Z:	Valor de Z



```
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=23
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-9,7,-300,300
```



```
40 DEF FNf(x)=COS(4*x*x+z*z)*((x*x+z*z)/(x*z+1.1))+ABS(SIN(1
0*SQR(x*x)))
50 ap=24
80 DATA -4,4,-12,12,-4,4
```

3. Programas BASIC

3.1 Generador de menú

La mayor parte de programas extensos prevén, que el usuario pueda controlarlos ayudado por un menú. Actualmente se tiende a configurar los programas de forma que su manejo sea sencillo. Estamos seguros, de que esta tendencia se mantendrá en un futuro, y que posiblemente se agudice aún más.

Una condición importante para el fácil manejo de un programa es la de tener un menú confortable. Con ayuda del menú, el usuario puede elegir, al igual que en la carta de un restaurante, lo que desea comer. En nuestro caso, se trata de elegir una función del programa. Como la elección en una carta con muchas posibilidades puede ser difícil, o como mínimo necesita mucho tiempo, el contenido de la carta del menú se encuentra dividido en apartados.

Un menú principal nos lleva a submenús, los cuales asimismo tienen también sus apartados, hasta llegar al punto deseado. Además es aconsejable, que un buen menú también indique todas las funciones de servicio que no sean lo suficientemente claras, p.e. cómo llegar al menú anterior o posterior.

Como la programación de un menú no siempre es necesaria, hemos escrito un generador de menú, que comporta un ahorro de tiempo considerable.

Con un gasto mínimo, sólo cambiar las líneas DATA, donde se encuentran los puntos del menú, puede activarse el programa de forma universal.

El programa, sin el programa principal correspondiente, que es dirigido por el menú, carece de valor. El programa principal es siempre el suyo propio, al que se le añade el menú.

La técnica del generador de menú presenta una muy interesante posibilidad de aplicación de las variables de campo del BASIC. Sin ello, no sería posible escribir un programa. El fin de todo ello es presentar cada menú mediante una sola rutina. Después de que el usuario ha encontrado la elección, la rutina debe encontrar el camino automáticamente el el nuevo menú.

Todos los puntos del menú posibles están almacenados en una campo (m$), uno detrás de otro. Para retener dentro de este campo los puntos concretos del menú en un orden, existe un segundo campo (md para Menú DATA), donde para cada menú se encuentran almacenadas las siguientes informaciones:

- El número del menú siguiente
- El número del último menú
- La magnitud (número de puntos) de un menú
- La última elección encontrada en el último menú

La variable mepo contiene el número del menú actual. De esta forma, m$ (mepo) contiene al mismo tiempo el nombre de la parte del programa, en la cual nos hallamos en ese momento.

md(mepo,gros) contiene el número de puntos del menú actual, md(mepo,dflt) contiene el punto del menú, que se ha elegido en última instancia.

En md(mepo,ruck) se encuentra almacenado el número del menú, que puede conseguirse pulsando la tecla DEL (o sea, el último menú).

Finalmente el número del menú siguiente se encuentra almacenado en el primer punto del menú de un menú en md(mepo,nxt). md(mepo,nxt) tiene aún otra función en base a la desarrollada estructura de datos. Por una parte, como ya se ha dicho, se puede encontrar el número del menú siguiente, por otra, este valor es a la vez el número del primer punto del menú actual (ver líneas 520-560).

La parte más complicada del programa es el bucle, que consta de líneas DATA, en las que se encuentran los puntos del menú, y que activa y busca el puntero antes descrito. Las líneas DATA deben contener los puntos concretos del menú, donde cada menú se encuentra separado de los demás mediante un número correlativo. Allí se ofrece el menú principal después de 0, y luego para cada punto del menú principal, el submenú correspondiente en el orden necesario.

Los submenús para los primeros submenús se llevan a cabo de este modo. Si no existe un submenú para cada punto de selección de un menú, debe saltarse por encima de lo que falta en la numeración.

El bucle de inicialización (líneas 10100-10130) lee luego el punto de menú siguiente. Para cada punto leído, el puntero de salto de retorno se activa para el menú conseguido mediante la selección de este punto, donde se almacenará bajo md (i,ruck) el número del menú actual n. El valor de defecto (dflt), que contiene el último punto seleccionado del menú, se colocará ahora en 1, o sea en el primer punto. Luego se comprobará si se han leído todos los puntos. Al final de las líneas DATA, debe haber un "!".

Si se han leído todos los puntos, se buscará el tamaño del último menú y se acaba el bucle de inicialización.

En la línea 10120 se examinará si se ha leído un número; si es así, se leerá inmediatamente el siguiente punto del menú en la línea 10110. Luego se comprobará que se ha leído un punto del menú, y después se leerá n con el nuevo número. El paso siguiente aparece de un modo ilógico.

Como número del menú que sigue al nuevo menú, se almacenará el contador actual de los puntos leídos. Examine si este valor representa verdaderamente el número del menú siguiente. Después se busca aún la magnitud de lo últimamente leído.

Después de ejecutar el bucle de inicialización completo se produce esquemáticamente la imagen siguiente:

Tras inicializar el programa, ésta puede utilizarse en cada programa. Para llamar la rutina de salida de menú puede utilizarse p.e. el formato siguiente:

```
1010 GOSUB 500:IF ein$=del$ THEN RETURN
1020 ON wahl GOSUB  ...Número de línea
```

En los números de línea a los que se ha saltado, se encuentran otras líneas del tipo de 1010 y 1020, que indican el submenú siguiente.

```
10 REM menu
20 MODE 2
30 GOSUB 1000 : REM Init
40 GOSUB 300 : REM Menu principal
50 CLS:END
60 REM Rahmen
70 CLS:PRINT"Generador de menus";m$(mepo)
80 PRINT STRING$(80,"-");
90 LOCATE 1,23:PRINT STRING$(80,"-");
100 PRINT"Pulse >ENTER< para eleccion de representacion inve
rsa"
110 PRINT"     o >DEL< para";exit$(-(mepo>0)-(mepo>1));
120 RETURN
130 REM Construir menu
140 GOSUB 60 :REM Marco
150 wahl=md(mepo,dflt)
160 LOCATE 1,5
170 FOR i=1 TO md(mepo,gros):IF i=wahl THEN PAPER 1:PEN 0
180 PRINT i;:PAPER 0:PEN 1:PRINT" - ";:IF i=wahl THEN PAPER
1:PEN 0
190 PRINT m$(i+md(mepo,nxt)-1):PAPER 0:PEN 1
200 NEXT
210 PRINT:PRINT:PRINT:PRINT"Su eleccion :";:PAPER 1:PEN 0:PR
INT wahl;:PAPER 0:PEN 1:LOCATE POS(#0)-3,VPOS(#0)
220 ein$=INKEY$:IF ein$="" THEN 220
230 IF ein$=del$ THEN RETURN
240 IF ein$=CHR$(13) THEN md(mepo,dflt)=wahl:mepo=md(mepo,nx
t)+wahl-1:RETURN
250 IF ein$=CHR$(241) OR ein$=CHR$(242) OR ein$=CHR$(32) THE
N wahl=wahl+1+(wahl=md(mepo,gros))*md(mepo,gros):GOTO 290
260 IF ein$=CHR$(240) OR ein$=CHR$(243) THEN wahl=-(wahl=1)*
md(mepo,gros)+wahl-1:GOTO 290
270 n=VAL(ein$):IF (n<1) OR (n>md(mepo,gros)) THEN 220
280 wahl=n
290 PAPER 1:PEN 0:PRINT wahl;:PAPER 0:PEN 1:GOTO 160
300 REM Menu principal
310 GOSUB 130:IF ein$=del$ THEN RETURN
```

```
320 ON wahl GOSUB 340,3000,4000,5000,6000
330 GOTO 310
340 REM al lado de 1
350 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck):RETURN
360 ON wahl GOSUB 370,2200,2300,2400,2500,2600:GOTO 350
370 REM debajo de 1
380 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck):RETURN
390 ON wahl GOSUB 2150,2160,2170:RETURN
1000 REM Init
1010 del$=CHR$(127)
1020 FOR i=0 TO 2:READ exit$(i):NEXT
1030 DATA "Fin de programa"
1040 DATA "Menu principal"
1050 DATA "Ultimo menu"
1060 DIM m$(100),md(100,3):ruck=0:nxt=1:gros=2:dflt=3:n=-1:i
=-1
1070 i=i+1:READ m$(i):md(i,ruck)=n:md(i,dflt)=1:IF m$(i)="'"
 THEN i=i-1:md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+1:RETUR
N
1080 IF LEN(m$(i))<3 THEN n=VAL(m$(i)):md(n,nxt)=i:i=i-1:IF
md(i,ruck)>=0 THEN md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+
1
1090 GOTO 1070
1100 RETURN
1110 DATA "Menu principal"
1120 DATA 0
1130 DATA "Imprimir"
1140 DATA "Presentar"
1150 DATA "Editar"
1160 DATA "Buscar"
1170 DATA "Servicio"
1180 DATA 1
1190 DATA "Normal","Proporcional","Justificar margen derecho
","Justificar margen derecho+Proporcional"
1200 DATA 6
1210 DATA "Ultimo menu","Solo para pruebas","etc."
1220 DATA "!"
```

3.2 Generador de máscaras de entrada

En relación con la confección de programas de fácil manejo, también es necesario proteger toda comunicación con el usuario contra errores de manejo.

Las comunicaciones del usuario básicamente se producen en todos los lugares del programa, donde el usuario puede variar la ejecución del programa con una entrada, ya sea mediante teclado, joystick, lápiz óptico u otros dispositivos de entrada. El generador de menú descrito en el capítulo anterior, está constituído de tal forma que todas las entradas no admisibles serán ignoradas. Con el generador de máscara de entrada, debe realizarse esta posibilidad en cualquier tipo de entrada. Desarrollaremos este programa con un ejemplo:

Se deben introducir p.e., nombre, dirección, población, teléfono y fecha en un programa de direcciones. Para cada entrada debe preverse un espacio concreto. Para la entrada de nombres se permiten sólo letras, para la del número de teléfono, sólo números, etc.

Los datos para una máscara de entrada deben darse mediante líneas DATA.
Son importantes las siguientes indicaciones:

- Comentario para la entrada (p.e. Nombre)
- Posición en pantalla del comentario
- Longitud del campo de entrada
- Cifra de reconocimiento para la cantidad de entradas permitidas

La entrada de una máscara con todos sus campos se cerrará en su totalidad, es decir, no se podrá corregir el primer campo de entrada, si ya nos encontramos en otro campo. Tampoco ha de ser posible efectuar una entrada que sobrepase la longitud prevista del campo.

Las teclas de cursor "hacia derecha" y "hacia izquierda", mueven el cursor dentro del campo. RETURN y "cursor hacia abajo", mueven el cursor hacia el siguiente campo, y "cursor hacia arriba" hacia el campo anterior. Pulsando la tecla COPY se aceptará la entrada completa.

```
10 MODE 2
20 GOSUB 820
30 GOSUB 80: REM Ir a buscar entrada
40 REM Entrada constante a disposicion
50 LOCATE 1,20
60 FOR i=1 TO anzfeld:PRINT ein$(i):NEXT
70 END
80 CLS:feldnum=1
90 FOR i=1 TO anzfeld
100 LOCATE x(i),y(i)
110 PRINT koment$(i);":";STRING$(l(i),"."):ein$=STRING$(l(i)
," ")
120 ein$(i)=STRING$(l(i)," ")
130 NEXT
140 x=x(feldnum):y=y(feldnum)
150 lm=l(feldnum):kenzif=ken(feldnum)
160 LOCATE x,y:PRINT koment$(feldnum);":";
170 x=x+LEN(koment$(feldnum))+1
180 IF x>80 THEN y=y+x\80:x=x MOD 80
190 e$=ein$(feldnum)
200 GOSUB 280
210 ein$(feldnum)=e$
220 IF endflg THEN RETURN: REM Entrada acabada
230 IF hochflg THEN feldnum=feldnum-1:IF feldnum=0 THEN feld
num=1
240 IF runtflg THEN feldnum=feldnum+1:IFfeldnum>anzfeld THEN
 feldnum=anzfeld
250 runtflg=0:hochflg=0
260 GOTO 140
270 PRINT CHR$(7);:GOTO 360
280 ' Ir a buscar entrada
290 ' x,y : Coordenada de entrada
300 ' lm :Longitud maxima del campo de entrada
310 ' aus :Window #
320 ' Vuelta :e$
330 LOCATE x,y
340 l=1
350 CALL &BB81: REM Cursor conectado
360 a$=INKEY$:IF a$="" THEN 360
```

```
370 a%=ASC(a$)
380 FOR i=1 TO anstz:IF a%<>stz%(i) THEN NEXT
390 ON i GOTO 470,470,510,540,570,590
400 REM Test para entrada permitida
410 okflg=-1
420 ON kenzif GOSUB 610,660,720,770
430 IF okflg=0 THEN 270
440 IF POS(#aus)>=x+lm THEN 270
450 e$=LEFT$(e$,POS(#aus)-x)+a$+RIGHT$(e$,lm-(POS(#aus)-x)-1
)
460 PRINT a$;:GOTO 360
470 REM Return y cursor hacia abajo
480 runtflg=-1
490 CALL &BB84: REM Cursor fuera
500 RETURN
510 REM chr$(242) Cursor derecha
520 IF POS (#aus)=x THEN 270
530 PRINT CHR$(8);:GOTO 360
540 REM chr$(243) Cursor izquierda
550 IF POS(#aus)>=x+lm THEN 270
560 PRINT CHR$(9);:GOTO 360
570 REM Cursor arriba
580 hochflg=-1:GOTO 490
590 REM copia
600 endflg=-1:GOTO 490
610 REM Test admite letras
620 IF a%>64 AND a%<91 THEN RETURN
630 IF a%>96 AND a%<123 THEN RETURN
640 IF a%=32 THEN RETURN
650 okflg=0:RETURN
660 REM Test admite letras y numeros
670 GOSUB 610
680 IF okflg THEN RETURN ELSE okflg=-1
690 REM Test para numeros
700 IF a%>47 AND a%<59 THEN RETURN
710 okflg=0:RETURN
720 REM Test admite telefono, o sea numeros y "/"
730 GOSUB 690
740 IF okflg THEN RETURN ELSE okflg=-1
750 IF a%=47 THEN RETURN
```

```
760 okflg=0:RETURN
770 REM Test admite fecha, o sea numeros y puntos
780 GOSUB 690
790 IF okflg THEN RETURN ELSE okflg=-1
800 IF a%=46 THEN RETURN
810 okflg=0:RETURN
820 REM Init
830 ' anstz : Numero caracteres de control
840 ' stz%(anstz) : valor ASCII de caracteres de control per
mitidos
850 anstz=6:FOR i=1 TI anstz:READ stz%(i):NEXT
860 DATA 13,241,242,243,240,224
870 aus=0:REM Numero de window (ventana)
880 REM Mascara de entrada
890 READ anzfeld : REM Numero de campos de entrada
900 FOR i=1 TO anzfeld
910 READ koment$(i),x(i),y(i):REM Comentario, posicion x e y
920 READ l(i): REM Longitud campo de entrada
930 READ ken(i):REM Cifra para entrada permitida
940 ' 1: Letra, 2:Letras y numeros, 3:Telefono, 4:Fecha
950 NEXT i
960 RETURN
970 REM Datos menu
980 DATA 5
990 DATA "Nombre   ",1,4,20,1
1000 DATA "Fecha    ",50,4,8,4
1010 DATA "Calle",1,6,20,2
1020 DATA "Poblacion   ",1,9,30,2
1030 DATA Telefono,50,9,12,3
```

Lista Cross-Reference creada con al instrucción XREF:

```
AUS ! 440 450 450 520 550 870
ANSTZ ! 380 850 850
A % 370 380 620 620 630 630 640 700 700 750 800
A $ 360 360 370 450 460
ANZFELD ! 60 90 240 240 890 900
ENDFLG ! 220 600
E $ 190 210 450 450 450
EIN $ 60 110 120 190 210
FELDNUM ! 80 140 140 150 150 160 170 190 210 230 230 230  230
240 240 240 240
HOCHFLG ! 230 250 580
I ! 60 60 90 100 100 110 110 110 120 120 380 390 850 850  900
910 910 920 930 950
KEN ! 150 930
KENZIF ! 150 420
KOMENT $ 110 160 170 910
LM ! 150 440 450 550
L ! 110 110 120 150 340 920
LIWST !
OKFLG ! 410 430 650 680 680 710 740 740 760 790 790 810
STZ % 380 850
X  ! 100 140 140 160 170 170 180 180 180 330 440 450 450  520
550 910
Y ! 100 140 140 160 180 330 910
```

4. Circle

El listado presentado aquí ofrece la posibilidad de simular la instrucción circle, que tampoco está incluída en las nuevas versiones de CPC, con extraordinaria rapidez desde el BASIC.

Puesto que la rutina sólo calcula un 1/4 de círculo, generando los cuartos restantes mediante las simetrias del valor obtenido, es más rápida que las rutinas circle "normales".

Llamando la rutina mediante GOSUB, R debe contener el radio, X e Y, la posición y e la excentricidad.

```
10 RAD
20 MODE 2
30 INPUT"Radio,Excentric. ",r,e
40 INPUT"Coor. X, Coor. Y ",xk,yk
42 GOSUB 60
43 END
60 x(0)=0:y(0)=r*e
70 x(1)=0:y(1)=-r*e
80 x(2)=0:y(2)=-r*e
90 x(3)=0:y(3)=r*e
100 FOR al=0 TO PI/2 STEP PI/18
110 x=r*SIN(al):y=SQR(r*r-x*x)*e
120 i=0:GOSUB 180
130 i=1:y=-y:GOSUB 180
140 i=2:x=-x:GOSUB 180
150 i=3:y=-y:GOSUB 180
160 NEXT
170 RETURN
180 MOVE xk+x(i),yk+y(i):x(i)=x:y(i)=y
190 DRAW xk+x(i),yk+y(i):RETURN
```

5. Proteger programas propios

Una posibilidad sencilla de proteger programas propios de miradas curiosas, consiste en almacenarlos con la opción "p". La "p" se separará mediante coma, añadida tras el nombre del programa.

```
SAVE "nombre.xxx",p
```

Un programa de este tipo puede iniciarse sólo con la instrucción RUN "nombre.xxx". Pero la ejecución del programa puede interrumpirse con la tecla ESC. De este modo no será posible efectuar un listado directamente, pero los genios tampoco tardarían en descubrirlo.

Si ocupamos el espacio de memoria &BDEE con el valor hexadecimal &C9

```
POKE &BDEE,&C9
```

podremos interrumpir el programa activando la tecla ESC. El mismo efecto se puede conseguir también con la instrucción BASIC KEY DEF:

```
KEY DEF 66,1,0     Activar tecla ESC
KEY DEF 66,1,252   Desactivar tecla ESC
```

Se sobreentiende que el programa debe estar realizado de tal modo, que no pueda ser bloqueado o forzado a un aviso de error a causa de una entrada falsa premeditada, lo que igualmente provocará una interrupción del programa.
Si un programa es capaz de examinar todas las eventualidades, y saber cuales conducen a un aviso de error, se puede llevar a cabo una interrupción mediante la instrucción

```
ON ERROR GOTO número de línea
```

El programador decidirá libremente si en la línea dada se ejecutará un aviso de error o, p.ej., un CALL 0.

Ejemplo:

```
10 CONTADOR=0
20 INPUT"Divisor",T
30 PRINT 10/T
40 ON ERROR GOTO 100
50 GOTO 20
100 CONTADOR=CONTADOR+1
110 IF CONTADOR<4 THEN PRINT"entrada falsa" ELSE CALL 0
120 GOTO 50
```

En este caso el usuario puede permitirse realizar una falsa entrada tres veces. Si efectúa una cuarta entrada erronea, mediante CALL 0 se provoca un RESET.

A un buen programa le corresponde un buen final. Para evitar, que el programa pueda ser descubierto después de este final, debe ser destruído dentro del ordenador; ésta es la mejor protección contra el copiado.
Por lo tanto, si la pregunta por el final del programa es contestada positivamente, simplemente hay que activar la instrucción CALL 0. Esta provocará un RESET, igual al efectuado con la combinación de las teclas CTRL/SHIFT/ESC.

6. Facilidades para la entrada de programas

Al entrar programas largos suele ocurrir, que se coloquen espacios (Blancos) innecesarios. Estos espacios ocupan memoria, que más adelante puede faltarle al usuario.
Pero es casi imposible, que al entrar los programas pongamos nuestra atención por un lado en la sintaxis, y por otro controlemos además los espacios que son imprescindibles y los que no lo son.

Pero podemos olvidarnos de este problema, si POKEamos un valor diferente de cero en la posición de memoria &AC00 antes de entrar el programa. En este caso, el ordenador controla los espacios que forman parte de la sintaxis, omitiendo todos los demás. Ello incluso nos permite,en caso de duda, entrar algunos espacios de más 'por si acaso'. Al final sólo se transmiten los absolutamente imprescindibles.

7. Aceleración de programas BASIC

Aunque el BASIC del Amstrad no deja mucho que desear en cuanto a la velocidad, existen problemas, en los que el factor tiempo juega un papel importante (p.e. el procedimiento de ordenación). Si un algoritmo no puede acelerarse más dentro de una versión, a menudo se pueden conseguir aumentos de velocidad mediante técnicas hábiles de programación. Ahora les presentaremos algunas de estas posibilidades.

Con la variable BASIC TIME, tenemos un instrumento simple para medir el tiempo de ejecución BASIC. Volvamos al procedimiento de ordenación de burbuja. Con este ejemplo queremos mostrar cómo se puede conseguir en general la aceleración de programas. En el siguiente programa hemos escrito una parte permanente y una parte de variable, para hacer más claro el ejemplo. Las líneas de 10 a 90 permanecerán siempre igual, no deben teclearse de nuevo.

```
10 REM BASIC Velocidad
20 anz=10:anz=anz-1
30 DIM Feldelement(anz)
40 FOR i=0 TO anz:READ Feldelement(i):NEXT
50 DATA 10,4,7,3,2,24,8,17,5,19
60 t!=TIME
70 GOSUB 100
80 PRINT (TIME-t!)/300
90 END
100 REM Ordenacion de burbuja
110 feldgros=anz : REM Primer campo a clasificar es campo to
tal
120 index=1 : REM Empieza comparacion con primeros elementos

130 IF Feldelement(index-1)>Feldelement(index) THEN GOSUB 19
0 : REM Intercambio
140    index=index+1 : REM Comparar elemento siguiente
150    IF index<=feldgros THEN 130 : REM Para comparar magni
tud
160 feldgros=feldgros-1 : REM Disminuir campo a ordenar
170 IF feldgros>=1 THEN GOTO 120 : REM Ordenar campo disminu
ido
180 RETURN
190 REM Subprograma : Intercambio de dos elementos
200 zwischenspeicher=Feldelement(index)
210 Feldelement(index)=Feldelement(index-1)
220 Feldelement(index-1)=zwischenspeicher
230 RETURN
```

Tiempo: 0.62 segundos con líneas REM

Líneas REM

La primera mejora de tiempo la conseguimos con la eliminación de las líneas REM o "'" (Shift 7).

Tiempo: 0.56 segundos sin líneas REM

Elección de variables

Uno de los puntos más importantes es la elección de variables. El BASIC conoce variables ENTERAS (INT) y variables REALES (REAL). Las variables REALES tienen una longitud de 5 bytes y para su tratamiento necesitan más tiempo que las variables ENTERAS, de sólo 2 bytes. Por ello, si es posible, es mejor elegir variables INT. En la mayoría de casos podemos renunciar a las variables REAL.

```
5 DEFINT a-z
100 feldgros=anz
120 index=1
130 IF Feldelement(index-1)>Feldelement(index) THEN GOSUB 19
0
140 index=index+1
150 IF index<=feldgros-1 THEN 130
160 feldgros=feldgros-1
170 IF feldgros>=1 THEN GOTO 120
180 RETURN
190 zwischenspeicher=Feldelement(index)
210 Feldelement(index)=Feldelement(index-1)
220 Feldelement(index-1)=zwischenspeicher
230 RETURN
```

Tiempo: 0.42 segundos sin variables REAL

Las variables INIT sólo pueden tomar valores de -32768 a 32767.

Por esta razón, nos encontramos con problemas al tratar con direcciones comprendidas entre 0 y 65535 (&0000 a &FFFF). Sin embargo, en lugar de utilizar ahora variables REAL, tenemos la opción de tratar todas las direcciones mayores que &7FFF como números negativos.

Bucles FOR-NEXT

Si se han de programar bucles, el bucle FOR-NEXT **es más** rápido por ejemplo que el bucle IF-THEN-GOTO.

Tiempo: 0.34 segundos con FOR-NEXT

Nombres de variables y definiciones

Cuanto más corto sea el nombre de la variable, más rápido es su tratamiento. Como la señal de impresión comporta también un espacio, deberíamos definir antes todas las variables con DEF INT/ REAL/ STR.

```
5 DEFINT a-z
10 REM BASIC Velocidad
20 a=10:a=a-1
30 DIM e(a)
40 FOR i=0 TO a:READ e(i):NEXT
100 FOR f=a TO 1 STEP -1
110 FOR i=1 TO f
120 IF e(i-1)>e(i) THEN GOSUB 150
130 NEXT i,f
140 RETURN
150 z=e(i)
160 e(i)=e(i-1)
170 e(i-1)=z
180 RETURN
```

Tiempo: 0.35 segundos con nombres cortos de variables predefinidos

Líneas-BASIC

En el tratamiento interno, se trata de modo más rápido en cualquier caso, una línea larga de BASIC, que algunas líneas cortas de programa, que no se utilicen como líneas de salto, y por ello, se deberían juntar con otras.

```
100 FOR f=a TO 1 STEP -1:FOR i=1 TO f:IF e(i-1)>e(i) THEN GO
SUB 120
110 NEXT i,f:RETURN
120 z=e(i):e(i)=e(i-1):e(i-1)=z:RETURN
```

Tiempo: 0.34 segundos con menos líneas de programa

Como puede observar hemos superado considerablemente la velocidad de la primera versión. Si no nos es suficiente aún, debemos pasar ya al lenguaje máquina.

Otros trucos que pueden demostrarse con el programa anterior.

- Antes de utilizar variables con la misma letra inicial, es preferible utilizar letras que todavía estén libres

- Si tenemos variables con la misma letra inicial, es conveniente, que las variables que aparecerán previsiblemente con mayor frecuencia en el programa, sean inicializadas en primer lugar en el mismo, aunque esto no sea necesario. Veamos un ejemplo:

  I se utiliza más a menudo que IN$. IN$ se encuentra en la línea 10, I en la línea 100. Por ello se debería introducir p.e. simplemente una línea 9 I=0.

Las primeras variables encontradas son las primeras que se colocan en la tabla de variables, y por ello también serán las primeras en ser halladas en adelante.

- En el cálculo, la operaciones de cálculo fundamentales son siempre mas rápidas. P.e. utilizar X*X en lugar de X^2. También es aconsejable colocar paréntesis sólo en casos absolutamente necesarios.

- No introduzca indicaciones innecesarias en los bucles. Si se debe elegir p.e. un color concreto para la ejecución del bucle, esto puede anteponerse al mismo.
  Las instrucciones que no necesitan un tiempo de ejecución, como REM o DATA, no pertenecen a un bucle del programa. Aunque no se deban "ejecutar", deben interpretarse cada vez, lo que también cuesta tiempo.

- Evite todos los espacios innecesarios del texto del programa. También deben ser interpretados y, por tanto, gastan tiempo.

- Si trabaja mucho con asignaciones de cadena, más de una vez podrá producirse una Garbage Collection. En esta "acción de recogida de basura", se borrarán de la memoria todos los valores que ya no sean necesarios, recuperando así el espacio ocupado. Esta acción suele durar como varios segundos, en los que no se puede influir sobre el ordenador. Es imposible evitar la Garbage Collection. Pero podemos efectuarla continuamente por partes, invirtiendo así sólo milisegundos cada vez.

La instrucción PRINT FRE("") provoca la Garbage Collection. Si la colocamos hábilmente, p.e. detrás de una instrucción INPUT, donde suele producirse un cierto retardo hasta la entrada, o si la llamamos a ciertos intervalos con la instrucción EVERY GOSUB, podemos evitar por completo la pérdida de tiempo provocada por la Garbage Collection.

8. Instrucciones especiales del BASIC de Amstrad

8.1 EVERY a,b GOSUB c

Con la instrucción EVERY-GOSUB disponemos de la fantástica posibilidad,de poder utilizar interrupciones llevadas a cabo por el Hardware, mediante instrucciones BASIC. Esto posibilita al programador poder escribir programas multitarea en lenguaje BASIC, sin tener que dar rodeos en el lenguaje máquina.
Con una hábil utilización de esta instrucción, el ordenador puede tratar más de un programa al mismo tiempo. Con ello pueden administrarse especialmente subprogramas críticos de tiempo, p.e. un contador de tiempo, que trabaja de forma óptima y con un gasto mínimo.
Si llamamos la instrucción EVERY con los parámetros correspondientes, que seguidamente aclararemos, el interpretador BASIC lo deja todo en su sitio, y se bifurca, con GOSUB a la parte del programa dada.

Antes de ofrecer unas aclaraciones más completas, presentamos el siguiente programa que ilustra una aplicación de esta instrucción.

```
5 MODE 2 : ZEIT=0
10 EVERY 50,0 GOSUB 100
20 FOR I=0 TO 2*PI STEP PI/360
30 PLOT 320+300*SIN(I),200+195*COS(I)
40 NEXT I
50 END
100 ZEIT=ZEIT+1
110 LOCATE 50,10
120 PRINT ZEIT
130 RETURN
```

Este ejemplo de la figura Lissajou, que ya hemos utilizado varias veces, es muy apropiado para la ilustración de la instrucción EVERY.
EVERY llama un UP, que incrementa y presenta un contador de tiempo.

Este UP debe ser llamado, naturalmente, en intervalos iguales. Como la duración del cálculo de las funciones SIN y COS, depende en cierto modo de los valores de las mismas, -así p.e. el cálculo del seno de PI/3 dura bastante más tiempo que el de PI/4- no se puede intercalar de un indicador de tiempo en este bucle PLOT.
Este indicador se coloca en el UP, que se dirige mediante la instrucción EVERY, y que lo llama a intervalos de tiempo iguales.

Los parámetros a, b, y c:

a - Indica el número de unidades de tiempo (0.02 seg.), tras las que debe llamarse el UP.

b - Es el número del indicador de tiempo elegido. Tenemos a nuestra disposición cuatro indicadores de tiempo con los números de 0 a 3, donde 3 tiene la máxima prioridad. Si se han incorporado en el programa más llamadas a UP, se tratarán según el rango. Si se produce p.e una llamada con prioridad 2, mientras se está tratando aún una ejecución de otra de rango inferior, la que tiene mayor prioridad se ejecuta primero, y luego le siguen las demás según su prioridad. Según esto, la prioridad debe estar delimitada exactamente.

c - Señala el número de línea del UP deseado. El sistema operativo se bifurca a este número de línea a intérvalos dados por a.

Los relojes con prioridad superior tienen necesariamente preferencia frente a los de prioridad inferior. Todos los relojes tienen un rango superior alprograma principal. Ello puede tener como consecuencia, que en casos de programación inadecuada y si la bifurcación provocada por el reloj necesita demasiado tiempo, no quede tiempo para el programa principal. El siguiente ejemplo lo ilustrará.

```
5 MODE 2
10 EVERY 30 GOSUB 100
20 FOR I=1 TO 639 : PLOT I,100 : NEXT I
50 END
100 REM SUBPROGRAMA
110 FOR J=1 TO 30 : LOCATE 20,5 : PRINT J : NEXT J
120 RETURN
```

El "subprograma" a partir de la línea 100 desperdicia tanto tiempo, que para el programa restante, los PLOTs de una línea, apenas queda tiempo libre para el cálculo.
Así pues, no deberíamos utilizar esta instrucción de este modo.

Para desarrollar plenamente la instrucción EVERY, debería leer también directamente las aclaraciones para las instrucciones DI y EI, que se describen en las indicaciones de la instrucción AFTER.

8.2 AFTER a,b GOSUB c

La instrucción AFTER provoca un salto condicional hacia un subprograma. El salto se efectuará después de un cierto tiempo.

Los parámetros:

a - Número de unidades de tiempo (0.02 seg.); después de ello, se efectúa la instrucción de salto.

b - Entrada del indicador de tiempo elegido, tenemos a disposición los tres indicadores de 0 a 3, donde 3 tiene un rango superior.

c - Señala el primer número de línea del UP.

A diferencia de la instrucción EVERY, que tiene siempre una llamada UP repetitiva, la instrucción AFTER produce un sólo salto al UP elegido.
Las instrucciones AFTER-GOSUB con prioridades mayores, en caso de interferencia temporal, interrumpen los UPs, que se habían llamado en las instrucciones AFTER-GOSUB con prioridad inferior. Estos se llevarán a cabo más tarde.

Las instrucciones EVERY y AFTER simulan por software un control de interrupción que se encuentra por hardware en el Z80. Así se encuentran también las instrucciones Z80 DI y EI en el BASIC del Amstrad.

DI significa Disable Interrupt (Desconectar interrupciones)
EI significa Enable Interrupt (Admitir interrupciones)

Si es necesario proteger un subprograma, que se ha llamado mediante EVERY o AFTER, contra interrupciones de mayor prioridad, podemos hacerlo con la instrucción DI. Esta impide la interrupción de otro indicador de tiempo, bien sea producido por EVERY o AFTER.
Así DI protege un subprograma, que se haya llamado p.e con EVERY 15,2 GOSUB, de una interrupción mediante AFTER 10,2 GOSUB. Debemos tener presente que en este ejemplo se ha utilizado el mismo indicador de tiempo.

Con la instrucción EI se admitirán de nuevo todas las interrupciones. La desconexión anterior de las interrupciones, no se ha llevado a cabo de forma absoluta. Las interrupciones efectuadas entretiempo, se habían almacenado, y mediante EI se han recuperado de nuevo.

Inicie para ello el siguiente programa:

```
10 EVERY 20,0 GOSUB 100
20 EVERY 10,2 GOSUB 200
30 GOTO 30
90 REM--------------------------------
100 DI
110 FOR I=1 TO 10 : PRINT I; : NEXT
120 PRINT : EI : RETURN
130 REM--------------------------------
200 PRINT"Interrupcion prioridad 2"
210 RETURN
```

Cambie el valor final del bucle FOR-NEXT de 10 a 50.
Se observa que el programa, protegido por DI, puede tratarse primero en su totalidad, antes de la bifurcación en la línea 200 provocada por la mayor prioridad. Luego el texto de la línea 200 se imprime no sólo una vez, sino una detrás de otra, tantas veces como exigencias de interrupción aparecieron durante el bucle FOR-NEXT.

8.3 La instrucción MOD

La potencia del ordenador Amstrad se basa en estas instrucciones, que substituyen largas secuencias de instrucciones. De este modo no sólo se reduce el código fuente, sino que además el contenido es más claro.

Vamos a presentar los códigos ASCII de 33 a 255 con sus correspondientes caracteres en cuatro columnas de la pantalla.

```
10 FOR I=33 TO 255
20 PRINT I;" ";CHR$(I),
30 IF I/4=INT(I/4) THEN PRINT
40 NEXT
```

Las líneas 10 y 40 constituyen el bucle. En la línea 20 se imprime la variable de bucle y el carácter correspondiente. La coma detrás de la línea PRINT provoca una salida, una detrás de la otra. Para que sólo aparezcan las cuatro columnas deseadas, después de cuatro PRINTs con coma debe aparecer uno sin ella, para que se produzca un retorno de carro. En la línea 30 se debe averiguar cada cuarto proceso mediante la función INT (parte entera). Pero con la función MOD (módulo) esto se consigue de modo más simple y claro, pues el resultado de la misma es el así llamado resto de la división.

```
 4 MOD 4 = 0
 6 MOD 4 = 2
10 MOD11 =10
```

Si el resto de la división por 4 es igual a 0, el resultado de la función es igual a 0, dándose entonces la condición para el retorno del carro.

Se puede corregir la línea 30 de la forma siguiente:

```
30 IF I MOD 4=0 THEN PRINT
```

PARTE 2

Ampliación de instrucciones y otros programas útiles en lenguaje máquina

9. Ayudas para el programador

9.1 La estructura de la memoria de variables

Ahora nos ocuparemos del tratamiento interno de las variables BASIC. Al final de este apartado encontrará la ampliación de las instrucciones DUMP y XREF. DUMP presenta todas las variables que empiezan por una letra concreta con su valor correspondiente. XREF (Referencias cruzadas, Crossreference) muestra, para cualquier tipo de nombre de variable, todas las líneas de programa BASIC que contienen esta variable. DUMP y XREF son ampliaciones de instrucciones que facilitan especialmente la programación y la consiguiente búsqueda de errores.

El Locomotive-BASIC dispone en relación con las variables, de unas características estupendas:

Como de costumbre, existen tres tipos stándar de variables: enteras (número entero), reales (coma flotante) y de string (cadena, o sea, datos alfanuméricos). De este modo es grata la posibilidad, de asignar desde el principio las variables que empiezan con una letra concreta a un tipo fijo. Esta posibilidad debería utilizarse plenamente (ver capítulo 1: Aceleración de programas BASIC).

Una posibilidad completamente nueva en este tipo de ordenadores es la de poder asignar los nombres de variables con hasta 40 caracteres significativos. Esto es, se distinguen las diferencias que existen entre todas las letras o números del nombre de la variable. Hasta este momento se tenían en cuenta sólo las dos primeras letras del nombre de la variable. Por una parte, se consigue con ello la posibilidad de utilizar un número prácticamente ilimitado de nombres de variables, y por la otra, lo que es casi más importante, las variables se pueden "llamar por su nombre". Podemos escribir, por ejemplo, canaldesalida en lugar de CS.

De todos modos no debemos olvidar que el incremento en la longitud del nombre provoca que los programas vayan más lentos.

Para almacenar los nombres de variables de una longitud distinta, los usuarios del Locomotiv BASIC han pensado en una serie de trucos. Con ello se eleva la velocidad de tratamiento por encima de la de los métodos hasta ahora utilizados, aunque dependa también de la longitud del nombre.

Entremos primero en las bases de tratamiento de variables del ordenador.

Para la administración de los valores de las variables, al final de cada programa BASIC, se dispone de una tabla de variables. En esta tabla se encuentran todas las variables con su nombre, tipo y valor. Si en la ejecución de una instrucción BASIC, p.e. en una fórmula, se encuentra un nombre de variable, se podrá leer en la tabla junto con su valor correspondiente.
Si se utiliza una variable por primera vez, se transmite instantáneamente a la tabla, pues no se encuentra aún en ella.

En algunos dialectos BASIC existe la función VARPTR (VARiableR PoinTeR) que entrega la dirección de una variable (puntero) en la tabla de variables. También existe esta función en el BASIC del Amstrad, aunque no se mencione en el manual. El puntero de variable de una variable puede determinarse en el Amstrad mediante @ nombre de variable.

El valor obtenido (el VARPTR) es la dirección en la tabla de variables, donde se encuentra el valor de la variable. El valor está almacenado de forma distinta para los tres tipos posibles.

Variables enteras

En las variables INT el valor consta simplemente de los bytes alto (high) y bajo (low) del número. En la dirección menor, o sea, en la que indica VARPTR, se encuentra siempre el byte bajo (LB).

```
A%=100
PRINT PEEK(@A%)+256*PEEK(@A%+1)
```

muestra de nuevo 100, el valor asignado. Si la variable contiene números negativos, el valor tomado con PEEK debe convertirse con UNT. Esto es necesario, ya que existen números INT con signo y sin signo. -256 es un número INT con signo anterior, &8100 un número sin signo anterior. El número decimal tiene al final el valor 20736. PRINT &8100 muestra pues -256.

Con la representación del signo anterior, que se utiliza siempre, excepto con los números hexadecimales, el bit 15 del número se interpretará como signo.

Variables reales

En este tipo se almacenarán los números en la representación exponencial. El número se representa de tal forma que ocupe sólo una posición antes de la coma. La posición real de la coma se ofrece mediante los exponentes:

$$54321=5.4321*10^4.$$

De todos modos, esto se llevará a cabo internamente en sistema binario y no decimal.

Para almacenar una variable real, se utilizarán cinco bytes. Los cuatro primeros bytes forman la mantisa, esto es, la parte númerica del valor, donde, como convenido, la coma se encuentra siempre en la segunda posición. El MSB (Most Significant Bit - el bit de valor mayor) del byte 4 representa el signo del número. Finalmente el exponente se encuentra en el byte 5. Para obtener el valor real del exponente, debe restarse 129 del byte 5, o dicho de otro modo:

El exponente se representa con offset 129 (desplazamiento).

Como ya hemos dicho, el valor se almacena como número binario. O sea, la mantisa se compone sólo de ceros y unos. Como la mantisa tiene una posición de coma exacta, y como los ceros a la izquierda no figuran como posición, se mantienen sólo las unidades. Esto significa que la posición anterior a la coma del número binario es siempre uno.

Por ello no es necesario almacenar también el uno; en su lugar, se almacenará el signo . Del mismo modo que en los números INT, el byte bajo tiene el valor inferior, también aquí los bytes con direcciones inferiores tienen valores menores. A los cuatro bytes de la mantisa los llamaremos m1 a m4. De esta forma se obtiene la mantisa mediante:

```
PRINT (m1+256*m2+256^2*m3+256^3*(m4 OR 128))/256^4
```

"OR 128" pone de nuevo a 1 el bit no almacenado. La división por 256^4 es necesaria para que aparezca también un número decimal con una posición anterior a la coma.

Podemos calcular el valor de un número real con ayuda de VARPTRs a través del siguiente pequeño programa.

```
100 a=13:'coma flotante analizada
110 ad=@a:'Direccion de a
120 m1=PEEK(ad):m2=PEEK(ad+1):m3=PEEK(ad+2)
130 m4=PEEK(ad+3):ex=PEEK(ad+4)
140 PRINT (1-2*SGN(m4 AND 128))*2^(ex-129)*(1+(m4 AND 127)+(
m3+(m2+m1/256)/256)/128)
```

Variables de cadena (Strings)

El último grupo de variables es el que permite almacenar datos alfanuméricos. Una cadena es internamente sólo una serie de bytes uno detrás de otro, que contienen el código ASCII de los caracteres correspondientes. Como las cadenas pueden variar dentro en longitud entre 0 a 255, el contenido verdadero (los códigos) no podrá almacenarse directamente en la tabla de variables. Para almacenar las cadenas existe un espacio de almacenamiento propio en la RAM (en el extremo superior de la RAM BASIC), donde se almacenan sólo cadenas de caracteres. La tabla de variables contiene pues sólo la dirección de inicio de la cadena en la tabla de cadenas y la longitud de la cadena. Los dos valores juntos forman el llamado descriptor de la cadena. El siguiente programa aclara este procedimiento.

```
100 INPUT a$
110 ad=@a$
120 i=PEEK(ad)
130 stad=PEEK(ad+1)+256*PEEK(ad+2)
140 FOR i=stad TO stad+i-1:PRINT CHR$(PEEK(i));:NEXT i
```

Con ello hemos reseñado todos los tipos de variables. Observemos ahora la estructura completa de la tabla de variables.

La tabla de variables

La dirección de inicio de la tabla de variables puede leerse con PEEK la memoria RAM utilizada por el sistema.
En la dirección &AE85/6 (664,6128:&AE68/9) se encuentra la dirección de inicio de la tabla de variables.
La tabla de variables se ha constituído según la siguiente estructura de datos:

Número de bytes	Significado
2	Dirección de encadenamiento
hasta máx.39	Nombre de la variable (sin última letra), y última letra del nombre de la variable +128
1	Tipo de variable
2, 3 ó 5	Valor de la variable
	Inicio de la siguiente entrada según la misma estructura.

Entremos ahora en el significado de la dirección de encadenamiento. El nombre de la variable se almacenará en principio como cadena, esto es, en la forma de su código ASCII. Desgraciadamente este principio no es general. Las cifras que se hayan tomado en el nombre de la variable no se representan por su código ASCII.

Para indicar el final del nombre de la variable, se activa el bit 7 del código para la última letra (o cifra), o dicho de otro modo, se suma 128 al código.
El tipo de la variable se indica mediante una cifra. De este

modo:

1 Integer (Entera)
2 String (Cadena)
4 Real

Como Vd sabe, no se distingue entre mayúsculas y minúsculas. Los nombres de las variables se almacenan siempre en mayúsculas. La eventual transformación en minúsculas se realiza borrando el bit 5 de los códigos. Pruébelo:

```
PRINT CHR$(ASC(b) AND NOT 2^5)
```

Mediante estas operaciones, todas las minúsculas se transforman en mayúsculas. Ahora si sólo hay una cifra en el nombre de la variable, esta función naturalmente no funciona. El resultado es un código de control, menor que 32. Antes de una salida de los caracteres se debe activar el bit 5 mediante OR &20. El nombre se dará luego en minúsculas con las cifras correctas. OR &20 provoca la transformación de mayúsculas a minúsculas.

La cifra de reconocimiento de tipos en la tabla de variables es igual a la longitud del valor de la variable menos 1 (2,3 ó 5 bytes).

El valor, cuya longitud, como hemos dicho, es 2, 3 ó 5 bytes, contiene, codificando de la manera anteriormente descrita el valor (para números) o la dirección del "valor" (para cadenas).

En base a esta estructura sería posible buscar una variable concreta en la tabla, comparando simplemente todas las variables con la variable buscada. Esto, especialmente en programas largos, dura demasiado tiempo.

Para encontrar el nombre de una variable dentro de la tabla, podemos acelerar extremadamente la búsqueda utilizando listas encadenadas.
Para hacer que esto se haga realidad, se utiliza la dirección de encadenamiento que se encuentra al principio de cada entrada.

El principio de administración de datos a través de listas encadenadas, es que cada juego de datos, en este caso la entrada de variables, tiene un puntero asignado, que señala el siguiente juego relevante de datos. Esta es la dirección de encadenamiento.

En el Amstrad, todas las variables que empiezan con la misma letra (!), se encuentran encadenadas de este modo. Con ello se reduce en unas 26 veces el tiempo de búsqueda para una variable, pues existen 26 listas encadenadas, independientes unas de otras (una para cada letra). Las listas encadenadas necesitan algo más de espacio. Pero de todas formas vale la pena.
Para administrar una lista encadenada, son necesarias dos informaciones adicionales:

> Primero es necesaria la dirección del primer y último elemento de la lista. Estos elementos no se encuentran "encadenados" con ningún otro, por ello deben almacenarse por separado. La dirección del último elemento no es necesaria en el método aquí utilizado, pues se supone que la dirección de encadenamiento con el valor 0 significa que se ha alcanzado ya el último elemento. La dirección del primer elemento de la lista de cada letra será almacenada siempre en el estado actual en la RAM.

La búsqueda de una variable se realiza pues según el siguiente esquema:

1) Tomar la primera letra del nombre
2) Tomar para cada letra la correspondiente dirección de inicio del primer elemento
3) Leer y almacenar la siguiente dirección de encadenamiento
4) Analizar la concordancia del nombre de la variable
5) Si no corresponde el nombre de la variable, seguir con la dirección de encadenamiento (si es distinta a 0) en el punto tres; si la dirección de encadenamiento es igual a 0, no se ha dado concordancia

6) Si los nombres de las variables coinciden, se ha encontrado

Puesto que debe ser posible desplazar la tabla de variables, ya que es desplazada p.e. en cada modificación del programa BASIC, las direcciones de encadenamiento siempre están almacenadas como diferencia a la dirección de inicio de la tabla.

El siguiente programa se ha constituído en base a la estructura de las listas encadenadas de la tabla de variables.

9.2 DUMP - Salida de todos los valores de las variables

DUMP muestra todas las variables, cuya primera letra es la dada, junto con sus valores.

Como DUMP se encuentra unido a RSX (ver capítulo RSX para información más exacta), hay que escribir "raya" (Shift+@) antes de escribir el comando DUMP.
Si se han de mostrar todas las variables, se puede pulsar simplemente RETURN. Para entrar rangos de letras concretos, se deberá introducir primero "'"(Shift+7). La entrada del rango de letras se escribe luego del mismo modo que para las instrucciones DEFINT, DEFREAL y DEFSTR. Una instrucción completa sería por ejemplo:

```
|DUMP'A-C,F-L,X,Y
```

El listado Assembler del programa sirve a la vez para la instrucción XREF. No se deje despistar por ello, y sáltese las líneas que corresponden a XREF. De ello hablaremos más adelante.

Para llevar a cabo las operaciones aritméticas internas, existen los llamados FACs (acumuladores de coma flotante). Como el cálculo con números reales supone el tratamiento de 5 bytes, el almacenamiento en registros ya no tiene ningún sentido. Es decir, todas las operaciones referidas a números siempre son efectuadas con los FACs.

El FAC es una zona de la RAM, que ocupa 6 bytes. El primer byte de FAC contiene el tipo de la variable allí almacenada en ese momento (dirección &BOC1 para 464). Los bytes siguientes contienen el valor en la representación correspondiente. El valor de reconocimiento del tipo en FAC corresponde exactamente al número de bytes necesarios para el almacenamiento del valor, o sea, 2 para INT, 3 para String, y 5 para Real.
El FAC se utiliza para imprimir el valor con la rutina "PRINT FAC".

El listado ensamblador está documentado expresamente de forma muy detallada, especialmente para posibilitar una lectura agradable a los principiantes del lenguaje máquina.

```
A000            10                ; XREF
A000            20                ; [Cross]-REFference
A000            30
; mostrar números de linea BASIC
A000            40
; donden se encuentren
A000            50                ; unas variables concretas
A000            60                ; Formato :
A000            70
; "|xref'<rango(s) de letras>
A000            80                ; ejemplo: "|xref'c,f,i,w-z
A000            90                ;
A000           100                ; DUMP
A000           110
; mostrar todas las variables
A000           120                ; definidas y su valor
A000           130                ; Formato :
A000           140
: "|dump'<rango(s) de letras>
A000           150                ; ver ejemplo anterior
A000           160                ;
A000           170
; Salida impresora : POKE &ac06,8
A000           180                ; (464: POKE &ac21,8) directo
A000           190                ; antes de instrución
A000           200
A000           210                ; lazar con RSX
A000 010000    220  DEFRSX LD   bc,tabrsx
A003 210000    230         LD   hl,kernal
A006 CDD1BC    240         CALL &bcd1 ; extensión
A009 3EC9      250         LD   a,&c9 ; RET
A00B 3200A0    260         LD   (defrsx),a
; Impedir redefiniciones
A00E C9        270         RET
**** Linea 220 :  TABRSX=&A00F
A00F 0000      280  TABRSX DW   table
A011 C30000    290         JP   xref
A014 C30000    300         JP   dump
```

```
**** Linea 280   : TABLE=&A017
A017 585245   310   TABLE  DM    "XRE"
A01A C6       320          DB    &c6 ; "F"+&80
A01B 44554D   330          DM    "DUM"
A01E D0       340          DB    &d0 ; "P"+&80
A01F 00       350          DB    0
**** Linea 230 :  KERNAL=&A020
A020          360   KERNAL DS    4
A024          370
**** Linea 290 :  XREF=&A024
A024 3E01     380   XREF   LD    a,1
A026 320000   390   LLAMAR LD    (prgken),a
A029 DF       400          RST   &18 ; Solicitud con Call Far
A02A 0000     410          DW    vector ;ya que la ROM alta debe
A02C C9       420          RET    ; ser conectada
A02D          430
**** Linea 410   : VEKTOR=&A02D
A02D 0000     440   VEKTOR DW    start
A02F FD       450          DB    253 ; LROM off / UROM on
A030          460
**** Linea 300 :  DUMP=&A030
A030 3E00     470   DUMP   LD    a,0
A032 18F2     480          JR    llamar
A034          490
A034          500
; CPC       6128    464    664
A034          510   ALBAPC EQU   &ae58
; &ae75 , &ae58 memoria temporal para BASIC PC
A034          520   BASPC  EQU   &ae1d
; &ae36 , &ae1d Original BASIC PC
A034          530   TESBUC EQU   &ff92
; &ff71 , &ff92 Test para letra y UPPER
A034          540   SYNTER EQU   &d0d7
; &d07b , &d0da  presentar Syntax error
A034          550   GTVPTR EQU   &d619
; &d5db , &d61c leer puntero de tabla de variables
A034          560   BADOBC EQU   &e9b9
; &e8ff , &e9be Recorrer programa Basic y saltar a  dirección
BC
```

```
A034           570  CHKBRK EQU  &c472
; &c43c , &c475 comprobación de Break
A034           580  LNFEED EQU  &c398
; &c34e , &c39b Salida de Line Feed
A034           590  SKPCMD EQU  &e9fd
; &e943 , &ea02  Saltar Comando
A034           600  VAINIT EQU  &d6ec
; &ddb3 , &d6ef Introducir variable en tabla de variable
A034           610  CHRGET EQU  &de2c
; &dd3f , &de31 Leer siguiente byte
A034           620  CHRGOT EQU  &de37
; &dd4a , &de3c lee último byte
A034           630  CHRNEX EQU  &de25
; &dd37 , &de2a Comprobar byte siguiente
A034           640  CHKKOM EQU  &de41
; &dd55 , &de46 Comprobar la coma
A034           650  PRINT  EQU  &c3a0 ; &c356 , &c3a3
A034           660  PTLNNU EQU  &ef44
; &ee79 , &ef49 Imprimir número de línea
A034           670  VARFAC EQU  &ff6c
; &ff4b , &ff6c Copia Variable en FAC
A034           680  PRTFAC EQU  &f2d5
; &f236 , &f2da print FAC
**** Linea 390  : PRGKEN=&A034
A034           690  PRGKEN DS   2
A036           700  WRTADR DS   2
A038           710
**** Linea 440  : START=&A038
A038 3A34A0    720  START  LD   a,(prgken)
A03B FE00      730         CP   0 ; Dump ?
A03D 28FE      740         JR   z,sigue
A03F 010000    750         LD   bc,initva
; introducir todas las variables
A042 CDB9E9    760         CALL badobc
; Búsqueda en el programa BASIC
**** Linea  740 :  SIGUE=&A045
A045 2A58AE    770   SIGUE=LD   hl,(albapc)
A048 0641      780         LD   b,&41 ; "A" Comienzo (defecto)
A04A 0E5A      790         LD   c,&5a ; "Z" Fin (defecto)
```

```
A04C CD37DE   800           CALL chrgot ; Final ??
A04F B7       810           OR   a ; si es así,
A050 28FE     820           JR   z,inicio
; empezar con valor de defecto
A052 23       830           INC  hl ; PC en "'"
A053 CD25DE   840           CALL chrnex
; Analizar caracter siguiente
A056 C0       850           DB   &c0 ; El carácter debe ser "'"
A057 7E       860    VONVOR LD   a,(hl) ; Leer letras
A058 CD92FF   870           CALL texbuc
A05B 38FE     880           JR   c,ok ; Letra es ok
A50D C3D7D0   890    ERROR  JP   synter
**** Linea 880   : OK=&A060
A060 47       900    OK     LD   b,a ; Letra es valor de inicio
A061 4F       910           LD   c,a ; y también valor defecto
A062 CD2CDE   920           CALL chrget
A065 FE2D     930           CP   &2d ; "-" ??
A06720FE    940           JR   nz,fangan
A069 CD2CDE   950           CALL chrget
A06C CD92FF   960           CALL tesbuc
A06F 30EC     970           JR   nc,error
; si no hay letra
A071 4F       980           LD   c,a ; Es última letra
A072 CD2CDE   990           CALL chrget
**** Linea 940   : FANGAN=&A075
A075 E5       1000   FANGAN PUSH hl ; Salvar PC
A076 CD0000   1010          CALL inicio
A079 E1       1020          POP  hl ; PC
A07A CD41DE   1030          CALL chkkom ; comprobar coma
A07D 38D8     1040          JR   c,vonvor
A07F 2258AE   1050          LD   (albapc),hl
A082 C9       1060          RET
A083          1070
**** Linea 820   : INICIO=&A083
**** Linea 1010   : INICIO=&A083
A083 79       1080   INICIO LD   a,c ; Final
A084 90       1090          SUB  b ; menos inicio
A085 38D6     1100          JR   c,error
; Si fin<inic, entonces Syntax Error
```

```
A087 78       1110  NEXBUC LD   a,b
; XREF para variables con letras siguientes
A088 04       1120         INC  b ; para proxima vez
A089 C5       1130         PUSH bc ; aumentar y salvar
A08A CD19D6 1140           CALL gtvptr
; toma dirección del puntero de las letras
A08D 7E       1150  GLEBUC LD   a,(hl) ; HL se carga
A08E 23       1160         INC  hl ; con diferencia del inicio
A08F 66       1170         LD   h,(hl) ; de tabla de variables
A090 6F       1180         LD   l,a ; a la variable en cuestión
A091 B4       1190         OR   h
; si la diferencia es 0, no hay ninguna
A092 20FE     1200         JR   nz,namaus
; variable con el actual
A094 C1       1210         POP  bc
; letra de incio tomada
A095 79       1220         LD   a,c ; estado actual (b)
A096 B8       1230         CP   b ; comparar con final (c)
A097 30EE     1240      JR   nc,nexbuc
; Si aún no es final, intentar con letra siguiente
A099 C9       1250      RET    ; sino, listo
A09A          1260
**** Linea 1200  : NAMAUS=&A09A
A09A 09       1270  NAMAUS ADD  hl,bc
; incio + diferencia
A09B E5       1280         PUSH hl
; dirección de dirección de encadenamiento
A09C C5       1290         PUSH bc ; salvar inicio tab. Var.
A09D 23       1300         INC  hl ; encadenamiento
A09E 23       1310         INC  hl ; salto
A09F 7E       1320  AUSGA  LD   a,(hl) ; leer letras
A0A0 23       1330         INC  hl
A0A1 F5       1340         PUSH af
A0A2 E67F     1350         AND  &7f
; Bit 7 borrar para salida
A0A4 FE20     1360         CP   32
; número en nombre de variable?
A0A6 30FE     1370         JR   nc,allkla
; no, luego perfecto
```

```
A0A8 F620    1380           OR    &20 ; activar Bit 5 en números
**** Linea 1370   : ALLKLA=&A0AA
A0AA CDA0C3 1390  ALLKLA CALL print
A0AD F1      1400           POP   af ; valor leido
A0AE 17      1410           RLA   ; examina si bit 7 está activo.
A0AF 30EE    1420           JR    nc,ausga
; no, luego sigue edición
A0B1 3E20    1430           LD    a,&20 ; espacios vacios
A0B3 CDA0C3 1440           CALL print
A0B6 7E      1450           LD    a,(hl) ; Cifra identif. tipos
A0B7 23      1460           INC   hl
A0B8 C601    1470           ADD   a,1
; número de bytes=tipo para FAC
A0BA CD0000 1480           CALL typtes
A0BD F5      1490           PUSH af
A0BE 3434A0 1500           LD    a,(prgken)
A0C1 B7      1510           OR    a
A0C2 CA0000 1520           JP    z,fordum
; con DUMP, continuación
A0C5 F1      1530           POP   af
A0C6 C1      1540           POP   bc
A0C7 B7      1550           OR    a
A0C8 ED421560          SBC   hl,bc ; dirección de la Var.
A0CA 2236A0 1570           LD    (wrtadr),hl ; valores para
A0CD C5      1580           PUSH bc
; almacenar posterior comparación
A0CE 010000 1590           LD    bc,suchva
; rutina de búsqueda de variables
A0D1 CDB9E9 1600           CALL badobc
; recorrer programa BASIC
A0D4 CD72C4 1610  SOFERT CALL chkkbrk ; Break test
A0D7 CD98C3 1620           CALL lnfeed ; Line feed
A0DA C1      1630           POP   bc ; ini/fin letra
A0DB E1      1640           POP   hl ; direc. de encadenamiento
A0DC 18AF    1650           JR    glebuc
; siguiente variable con la misma letra
A0DE         1660
A0DE         1670                 ; rutina init de variable
**** Linea 750   :INITVA=&A0DE
```

```
A0DE E5      1680  INITVA PUSH hl ; PC
A0DF CDFDE9 1690          CALL skpcmd
; lectura de una instrucción
A0E2 D1      1700         POP  de
A0E3 FE02    1710         CP   2 ; Fin?
A0E5 D8      1720         RET  c ; si, linea siguiente
A0E6 FE0E    1730         CP   &e ; Variable ??
A0E8 30FA    1740         JR   nc,initva
; no, luego seguir buscando
A0EA FE07    1750         CP   7
A0EC 28F0    1760         JR   z,initva
A0EE FE08    1770         CP   8
A0F0 28EC    1780         JR   z,initva
A0F2 EB      1790         EX   de,hl
A0F3 D5      1800         PUSH de ; PC después de skip
A0FA CD2CDE 1810          CALL chrget
A0F7 CDECD6 1820          CALL vainit
; búsqueda de nombres, introducir variables
A0FA E1      1830         POP  hl
A0FB 18E1    1840         JR   Initva
A0FD         1850
A0FD         1860               ; rutina búsqueda de variables
**** Linea 1590   : SUCHVA=&A0FD
A0FD E5      1870  SUCHVA PUSH hl ; PC
A0FE CDFDE9 1880          CALL skpcmd
; lectura de una instrucción
A101 D1      1890         POP  de
A102 FE02    1900         CP   2 ; Final ?
A104 D8      1910         RET  c ; si, linea siguiente
A105 FE0E    1920         CP   & ; Variable ??
A107 30F4    1930         JR   nc,suchva
; no, luego seguir buscando
A109 FE07    1940         CP   7
A10B 28F0    1950         JR   z,suchva ; no
A10D FE08    1960         CP   8
A10F 28EC    1970         JR   z,suchva
A111 EB      1980         EX   de,hl
A112 D5      1990         PUSH de
A113 CD2CDE 2000          CALL chrget
```

```
A116  23       2010           INC  hl ; HL en  la  dirección  de
inicio
A117 5E        2020           LD   e,(hl)
A118 23        2030           INC  hl
A119 56        2040           LD   d,(hl)
A11A 2A36A0 2050              LD   hl,(wrtadr)
; con actual
A11D B7        2060           OR   a
A11E ED52      2070           SBC  hl,de ; comparar direcciones
A120 E1        2080           POP  hl ; PC después Skip
A121 20DA      2090           JR   nz,suchva
; diferencia, luego seguir buscando
A123 E5        2100           PUSH hl ; Adr. Typ. Prog.
A124 2A1DAE 2110              LD   hl,(baspc) ; Leer PC BASIC
A127 7E        2120           LD   a,(hl)
A128 23        2130           INC  hl
A129 66        2140           LD   h,(hl)
A12A 6F        2150           LD   l,a
A12B CD44EF 2160              CALL ptlnnu ; Print line Number HL
A12E 3E20      2170           LD   a,&20
A130 CDA0C3 2180              CALL print
A133 E1        2190           POP  hl
A134 18C7      2200           JR   suchva
A136           2210
**** Linea 1520   : FORDUM=&A136
A136 F1        2220    FORDUM POP  af ; Continuar DUMp
A137 FE03      2230           CP   3
A139 20FE      2240           JR   nz,nostri
A13B 46        2250           LD   b,(hl) ; longitud de cadena
A13C 97        2260           SUB  a ; borrar acumulador
A13D B8        2270           CP   b ; longitud es 0 ???
A13E 28FE      2280           JR   z,skip ; no hay edición
A140 E5        2290           PUSH hl ; dirección del descrictor
A141 23        2300           INC  hl
A142 7E        2310           LD   a,(hl) ; Byte bajo
A143 23        2320           INC  hl
A144 66        2330           LD   h,(hl) ; Byte alto
A145 6F        2340           LD   l,a
A146 7E        2350    NESTCH LD   a,(hl)
A147 23        2360           INC  hl
```

```
A148 CDA0C3 2370         CALL print
A14B 10F9   2380         DJNZ nestch
A14D E1     2390         POP  hl
**** Linea 2280  : SKIP=&A14E
A14E 3E03   2400  SKIP   LD   a,3 ; longitud del descriptor
A150 18FE   2410         JR   skval
**** Linea 2240  : NOSTRI=&A152
A152 E5     2420  NOSTRI PUSH hl
A153 CD6CFF 2430         CALL VARFAC
; activar tipo  y VAR (HL)-> FAC
A156 E1     2440         POP  hl
A157 CDD5F2 2450         CALL prtfac ; print FAC
**** Linea 2410  : SKVAL=&A15A
A15A C3D4A0 2460  SKVAL  JP   sofert
A15D        2470
**** Linea 1480  : TYPTES=&A15D
A15D F5     2480  TYPTES PUSH af ; el tipo emite
A15E E5     2490         PUSH hl
; carácter correspondiente
A15F E607   2500         AND  7 ; tratar solo Bit 0-2
A161 EE27   2510         XOR  &27
A163 FE22   2520         CP   &22
A165 20FE   2530         JR   nz,ok1
A167 D601   2540         SUB  1
**** Linea 2530  : OK1=&A169
A169 CDA0C3 2550  OK1    CALL print
A16C 3C20   2560         LD   a,&20; " "
A16E CDA0C3 2570         CALL print
A171 E1     2580         POP  hl
A172 F1     2590         POP  af
A173 C9     2600         RET

Programa : xdum
Inicio : &A000   Fin : &A173
Longitud : 0174
0 Error
Tabla de variables
DEFRSX A000  TABRSX A00F  TABLE  A017 KERNAL A020
XREF   A024  LLAMAR A026  VEKTOR A02D DUMP   A030
```

ALBAPC	AE58	BASPC	AE1D	TESBUC	FF92	SYNTER	D0D7
GTVPTR	D619	BADOBC	E9B9	CHKBRK	C472	LNFEED	C398
SKPCMD	E9FD	VAINIT	D6EC	CHRGET	DE2C	CHRGOT	DE37
CHRNEX	DE25	CHKKOM	DE41	PRINT	C3A0	PTLNNU	EF44
VARFAC	FF6C	PRTFAC	F2D5	PRGKEN	A034	WATADR	A036
START	A038	SIGUE	A045	VONVOR	A057	ERROR	A05D
OK	A060	FANGAN	A075	INICIO	A083	NEXBUC	A087
GLEBUC	A08D	NAMAUS	A09A	AUSGA	A09F	ALLKLA	A0AA
SOFERT	A0D4	INITVA	A0DE	SUCHVA	A0FD	FORDUM	A136
NESTCH	A146	SKIP	A14E	NOSTRI	A152	SKVAL	A15A
TYPTES	A15D	OK1	A169				

Aún no hemos hablado de una particularidad de la tabla de variables.

También las funciones, que están definidas con DEF FN, se encuentran almacenadas en la tabla de variables. El valor de identificación de tipos de una definición de función es &41, &42 ó &44. Esto es, el bit 6 corresponde a la "función", el nibble bajo (1, 2 ó 4) indica, del modo que ya nos es conocido, de qué tipo es el resultado de la función.

Todas las funciones están unidas entre sí, mediante la función de encadenamiento. En la dirección &AE04/5 (664/6128: &ADEB/C) se encuentra la dirección del primer elemento de la lista. El "valor" del nombre de una función consta de 2 bytes que indican el lugar del programa BASIC, donde se encuentra la definición de la función.

Las direcciones RAM del primer elemento para cada primera letra se encuentran en las direcciones de &ADD0 a &AE03 (para el 664 y 6128: de &ADB7 a ADEA), donde se utilizan 2 bytes para cada letra. Al utilizar este puntero, tenga siempre en cuenta, que no está indicando la dirección en sí, sino la diferencia entre esta dirección y la dirección de inicio de la tabla de variables &AE85/6 (664/6128: &AE68/9).

9.3 XREF (Cross REFerence)

La sintaxis de la instrucción XREF es la misma que la de la función DUMP. Como se utilizan grandes partes del programa tanto para DUMP como para XREF, se llevarán a cabo juntas en un mismo programa. Del mismo modo que con DUMP, las variables correspondientes se buscarán en la tabla mediante los bucles NEXBUC y GLEBUC.

Justo al principio del programa, XREF utiliza una rutina de sistema muy interesante para registrar todas las variables en la tabla de variables. En la dirección &E8FF (6128: &E9B9 / 664: &E9BE) se encuentra la rutina BADOBC, que ejecuta un programa BASIC línea por línea y, con la peculiaridad de que, después de hallado el principio de una línea, llama a otra rutina cualquiera, que puede analizar esta línea o algo por

el estilo.
La dirección de la rutina que debe realizar el análisis, se dará llamando BADOBC en el registro BC. En el programa XREF, es una vez la dirección de la rutina INITVA, y más tarde la de la rutina SUCHVA.

Para comprender la rutina SUCHVA debemos ocuparnos de la estructura de un programa BASIC, especialmente de la colocación de variables en el programa.

Antes de cada variable se encuentra una cifra de identificación. Dicha cifra indica, de qué tipo es la variable, y si se ha introducido el signo de identificación de tipos con ella (%, $ o !). Pueden utilizarse:

02 entera con %
03 cadenas con $
04 real con !
0B entera
0C cadena
0D real

A la cifra de identificación le sigue la diferencia entre la dirección de la variable en la tabla y la dirección de inicio de la tabla.
A este indicador de dirección le sigue directamente el nombre de la variable, donde, como siempre, se encuentra activado el bit 7 de la última letra.

SUCHVA comprueba en primer lugar, si se trata de una cifra de identificación. En caso afirmativo, se compara el nombre, y finalmente se cambia la cifra de identificación y también se compara. En caso de concordancia, se emitirá el número de línea BASIC de la línea actual.

Para programadores, que no poseen el Assembler del libro "Lenguaje máquina para CPC" u otro, ofrecemos seguidamente un cargador BASIC para el programa.

La unión de las nuevas instrucciones RSX se produce mediante CALL &A000. A partir de ese momento se encuentran DUMP y XREF en el correspondiente formato a su disposición.

```
10 ' XREF y DUMP para 464
20 ' RSX Activar con call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL ("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 49982 THEN PRINT"Error en datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA FC,A6,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,5F,03
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,FF,E8,2A,75,AE
170 DATA 06,41,0E,5A,CD,4A,DD,B7
180 DATA 28,31,23,CD,37,DD,C0,7E
190 DATA CD,71,FF,38,03,C3,7B,D0
200 DATA 47,4F,CD,3F,DD,FE,2D,20
210 DATA 0C,CD,3F,DD,CD,71,FF,30
220 DATA EC,4F,CD,3F,DD,E5,CD,83
230 DATA A0,E1,CD,55,DD,38,D8,22
240 DATA 75,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,DB,D5,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,56,C3,F1,17,30
```

```
300 DATA EE,3E,20,CD,56,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,FF,E8,CD,3C,C4,CD
350 DATA 4E,C3,C1,E1,18,AF,E5,CD
360 DATA 43,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,3F,DD,CD
390 DATA B3,D6,E1,18,E1,E5,CD,43
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,3F,DD,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,36,AE,7E
450 DATA 23,66,6F,CD,79,EE,3E,20
460 DATA CD,56,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,56,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,4B,FF,E1,CD
510 DATA 36,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,56,C3,3E,20,CD,56
540 DATA C3,E1,F1,C9
```

```
10 ' XREF y DUMP para 664
20 ' RSX Activar con call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50380 THEN PRINT"Error en datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
```

```
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 38,A0,3E,0D,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,E1,D0,18,D8
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,BE,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,3C,DE,B7
180 DATA 28,31,23,CD,2A,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,DA,D0
200 DATA 47,4F,CD,31,DE,FE,2D,20
210 DATA 0C,CD,31,DE,CD,92,FF,30
220 DATA EC,4F,CD,31,DE,E5,CD,83
230 DATA A0,E1,CD,46,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,1C,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A3,C3,F1,17,30
300 DATA EE,3E,20,CD,A3,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,BE,E9,CD,75,C4,CD
350 DATA 9B,C3,C1,E1,18,AF,E5,CD
360 DATA 02,EA,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,31,DE,CD
390 DATA EF,D6,E1,18,E1,E5,CD,02
400 DATA EA,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,31,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,49,EF,3E,20
460 DATA CD,A3,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
```

```
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A3,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA DA,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A3,C3,3E,20,CD,A3
540 DATA C3,E1,F1,C9
```

```
10 ' XREF y DUMP para 6128
20 ' RSX Activar con call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50608 THEN PRINT"Error en Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 20,A0,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,D1,00
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,B9,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,37,DE,B7
180 DATA 28,31,23,CD,25,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,D7,D0
200 DATA 47,4F,CD,2C,DE,FE,2D,20
210 DATA 0C,CD,2C,DE,CD,92,FF,30
220 DATA EC,4F,CD,2C,DE,E5,CD,83
230 DATA A0,E1,CD,41,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,19,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
```

```
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A0,C3,F1,17,30
300 DATA EE,3E,20,CD,A0,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,B9,E9,CD,72,C4,CD
350 DATA 9B,C3,C1,E1,18,AF,E5,CD
360 DATA FA,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,2C,DE,CD
390 DATA EC,D6,E1,18,E1,E5,CD,FD
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,2C,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,44,EF,3E,20
460 DATA CD,A0,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A0,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA D5,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A0,C3,3E,20,CD,A0
540 DATA C3,E1,F1,C9
```

10. Crear líneas BASIC desde el BASIC

Una función que hasta este momento falta en los dialectos BASIC, es la creación de una línea BASIC del programa.

Esta es pues la instrucción que se ofrece como nuevo aspecto en la programación de BASIC.

¿Existe una instrucción que posibilite la creación de líneas de programa desde el programa?

Si esto es posible, podemos pensar fácilmente en programas, que por su parte confeccionen programas, y que los programas así creados vuelvan a crear programas, etc. Así pues, ¿qué necesidad hay de aprender a programar, si pronto lo harán los mismos ordenadores?

Por suerte, no se ha llegado aún tan lejos. De todas formas, la "instrucción de creación de programas" ofrece posibilidades muy interesantes en este campo. Vayamos ahora a esta instrucción.

En la entrada de líneas BASIC en modo directo, estas se almacenan casi como una cadena en un buffer hasta pulsar la tecla RETURN. Luego se intenta interpretar la cadena como línea BASIC (reconocible por el número de línea que le precede). Si esto funciona, se transportará la cadena al formato interno de líneas BASIC. Allí se transformará, p.e. las instrucciones reconocibles en Tokens, y en esa forma, se almacenarán. Finalmente se insertará la línea BASIC así traducida, en base a su número de línea, dentro del programa en cuestión. Para ello, todas las líneas de mayor número se desplazarán previamente hacia arriba las posiciones correspondientes en la memoria.

Para poner en práctica nuestra idea, utilizamos las mismas rutinas, que se han utilizado en el procedimiento anterior por parte del interpretador. Unicamente necesitamos entregar al programa la dirección de la cadena (con @ variable cadena) que contiene la línea que se debe crear.

En la entrega de sólo un valor, éste se encuentra a nuestra disposición después de la llamada mediante CALL, o en las ampliaciones RSX en el registro DE. DE contiene la dirección del descriptor de cadena a trasladar. Después de comprobar la longitud de la cadena en el descriptor de cadena, se lee la dirección de la cadena y se carga en el registro HL. Con la rutina CHRSKP se saltan eventuales espacios vacíos y signos de movimiento del cursor (HT y LF). Si no se encuentra ningún byte nulo, se comprueba con TESTER, si existe un número al principio de la línea. Si es así, se traduce la línea con ASSEMB y se introduce. El final de la línea se puede reconocer gracias a los bytes nulos. A continuación tenemos el listado Assembler y el cargador BASIC correspondiente del programa descrito en ese caso.

```
A000            10                ; Liner
A000            20                ; crea lineas BASIC
A000            30                ; 1. en modo directo
A000            40                ; 2. en el programa, si
A000            50                ; NR. linea > linea actual
A000            60                ; a$ = linea en codigo
A000            70                ; ASCII, Final= Byte nulo
A000            80                ; Formato : call &a0000,@a$
A000            90
A000           100          ORG &a000
A000           110
;    CPC     6128 ; 464    ,   664
A000           120   CHRSKP EQU &de4d ; &dd61 , &de52
A000           130   TESTER EQU &eecf ; &ee04 , &eed4
A000           140   ASSEMB EQU &e7a5 ; &e6c6 , &e7aa
A000 DF        150          RST &18 ; Far Call
A001 0000      160          DW  vektor; pues BASIC ROM
A003 C9        170          RET    ; debe estar conectada
**** Linea 160  : VEKTOR=&A004
A004 0000      180   VEKTOR DW   start
A006 FD        190          DB   253 ; low ROM off, upp ROM on
**** Linea 180  : START=&A007
A007 EB        200   START  EX   de,hl
; dirección entregada con HL
A008 23        210          INC  hl ; leer extensión de cadena
A009  5E        220          LD   e,(hl) ; Lo Byte dirección  de
cadena
A00A 23        230          INC  hl
A00B  56        240          LD   d,(hl) ; Hi Byte dirección  de
cadena
A00C  EB         250          EX   de,hl ; dirección  de  cadena
tras HL
A00D CD4DDE   260          CALL chrskp
A010 B7        270          OR    a
A011 C8        280          RET  z
A012 CDCFEE   290          CALL tester
A015 D0        300          RET  nc
A016 CDA5E7   310          CALL assemb
; pasar linea e introducir; HL debe indicar
 el primer Byte de li̇nea ASCII
A019 C9        320          RET  ; Listo
```

```
Programa : liner
Inicio : &A000    Fin : &A019
Longitud : 001A
0 Error
Tabla de variables :
CHRSKP DE4D  TESTER EECF  ASSEMB E7A5  VEKTOR A004
START  A007
```

```
10 REM BASIC Cargador para Liner
20 FOR i=&A000 TO &A019
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 4275 THEN PRINT"Error en DATAs":END
60 '464: IF s<> 4123 THEN .....
70 '664: IF s<> 4290 THEN ....
80 PRINT"ok!":END
90 DATA DF,04,A0,C9,07,A0,FD,EB
100 DATA 23,5E,23,56,EB,CD,4D,DE
110 '664:..,..,..,..,..,..,52,DE
120 DATA B7,C8,CD,CF,EE,D0,CD,A5
130 '464:..,..,..,4,EE,..,..,C6
140 '664:..,..,..,D4,EE,..,..,AA
150 DATA E7,C9
160 '464:E6,C9
170 '664:E7,C9
```

Para demostrar más claramente el funcionamiento de la instrucción, presentamos a continuación un pequeño programa.

```
10 MEMORY &9FFF: REM Esta cargado el Liner?
20 ZEINU=100
30 Z$=STR$(ZEINU)+"REM Esta es la linea"+STR$(ZEINU)
40 Z$=Z$+CHR$(0):REM Caracter de identificacion final
50 CALL &A000,@Z$
60 ZEINU=ZEINU+10
70 IF ZEINU<210 THEN 30
80 Z$=STR$(ZEINU)+"LIST"+CHR$(0)
90 CALL &A000,@Z$
```

Limitaciones de este programa:

- El número de línea para las líneas a crear, debe ser mayor que el número de línea, en la que se ejecuta la instrucción CALL.

- La instrucción no se puede encontrar dentro de un bucle FOR-NEXT, ni de un bucle WHILE-WEND.

Es imprescindible tener en cuenta estas indicaciones, de otro modo, la ejecución del programa no volvería a la posición correcta y/o las variables no podrían ser encontradas de nuevo. Ello se debe al desplazamiento del programa BASIC para la inserción de la nueva línea.

La aplicación más simple de esta instrucción es seguramente la creación de líneas DATA para p.e. la administración de datos. Con ayuda del creador de datos, los datos pueden almacenarse directamente en el programa.

En el libro "Lenguaje máquina del CPC 464/664/6128" de DATA BECKER encontrará un programa, que utiliza la instrucción para la creación de un cargador BASIC a partir de código máquina que se encuentra en la memoria. Una aplicación profesional de la instrucción, siguiendo estos principios, podría ser más o menos:

Una empresa de Software distribuye programas de contabilidad. Para no tener que realizar un programa distinto para cada cliente, con sus condiciones específicas, se ha escrito un programa de desarrollo de Software, que prevé todas las posibilidades de acoplamiento. Después de entrar los deseos especiales, el sistema de desarrollo crea un programa individual adaptado en gran medida para cada caso.

11. Hardcopy de gráfico

Para que con la función plotter 3-D pueda efectuar también los dibujos en papel, le presentamos ahora un programa de hardcopy. El programa se ejecuta sin ninguna variación en la impresora EPSON FX-80 y otras compatibles. Para la adaptación a otros tipos de impresora, debe cambiarse sólo la secuencia de control, que conecta la impresora en el modo gráfico, teniendo siempre en cuenta que existen 8 puntos en el modo de patrón de bits.

Para la Hardcopy se leerá directamente la RAM de vídeo del Amstrad en la dirección &C000 - &FFFF en el modo 2. La memoria de pantalla del CPC está constituída de una manera peculiar, ya conocida. Pero siempre sucede, al igual que en casi todos los demás ordenadores, que un byte de la memoria de pantalla corresponde en la pantalla a un número determinado de puntos (en MODE 2 son 8 puntos). El problema se encuentra sólo en que la impresora imprime siempre 8 puntos uno debajo de otro, y no del modo que se encuentra en la memoria de la pantalla, uno al lado del otro.

Tratemos como ejemplo los siguientes signos especiales. Estos signos deben ser reproducidos mediante la impresora.

										Códigos en la memoria de pantalla
	0	0	0	0	0	0	0	0	=	00
	0	0	1	0	1	0	0	0	=	40
	0	1	0	0	0	1	0	0	=	68
	1	0	1	1	1	0	1	0	=	186
	0	1	0	0	0	1	0	0	=	68
	0	0	1	0	1	0	0	0	=	40
	0	0	0	1	0	0	0	0	=	16
	1	1	1	1	1	1	1	0	=	254
Códigos de impresora:	17	41	85	19	85	41	17	0		

Este carácter se reproducirá en pantalla mediante:

```
MODE 2
FOR I=&C000 TO &F8000 STEP &800
POKE I,A:NEXT
DATA 16,40,68,16,68,40,16,254
```

Para la salida en la impresora, los valores deben estar situados en forma de columnas.

```
PRINT#8, CHR$(27);"*"CHR$(1);CHR$(8);CHR$(0);
FOR I=0 TO 7:READ A
PRINT#8,CHR$(A);:NEXT
DATA 17,41,85,19,85,41,17,0
```

La primera instrucción PRINT#8 es la secuencia de códigos de control para la EPSON FX-80, que avisa la salida de 8 códigos de gráfico de patrón de bits.

El programa Hardcopy debe transformar la memoria total de pantalla paso a paso, para poder emitir las "columnas de valores". Como esto en lenguaje BASIC podría durar algunas horas, es más lógico utilizar en este caso el lenguaje máquina.

El programa Hardcopy utiliza algunas rutinas de sistema. Seguidamente vamos a aclararlo.

Para el Amstrad, el programa Hardcopy sin cambios en el hardware es quizás un poco más complicado. La causa de ello es que el creador del Amstrad que ha diseñado la interface Centronics para la impresora, sólo ha previsto 7 líneas de bits de datos. Normalmente las líneas son 8, ya que un byte contiene 8 bits y la impresora recibe byte a byte.

Mientras sólo se ocupe de imprimir textos y listados, esta limitación no tiene importancia. En el caso del Hardcopy, el bit que falta provoca una línea blanca en la imagen creada, que se encuentra cada ocho líneas de puntos.

El programa debe compensar esta deficiencia, haciendo que se impriman sólo siete líneas de puntos, y que se cree sólo un salto de línea después de una longitud de siete puntos. Como la memoria de pantalla del Amstrad está organizada en una dirección vertical y en unidades de 8 bytes, y como cada carácter tiene una altura de ocho bits, son necesarios una serie de cálculos para imprimir sólo siete bits a una velocidad considerable.

La pantalla dispone de 25 líneas para cada serie de ocho puntos, o sea, en total, 200 series de puntos. Si se elaboran sólo siete puntos de línea, deben transmitirse 28*7 series de puntos, y el problema aparece cuando al final nos sobran 4 líneas de puntos (200-28*7). Estas deben tratarse por separado. Al final, la pantalla tendrá 640=&280 puntos en posición horizontal.

Al conectar el modo de pantalla, se comunicará al FX-80 (y compatibles) la cantidad de bytes de gráfico que deben recibirse. El byte bajo de &280 es &80. En este byte se encuentra activado el octavo bit, por lo que este valor no se puede transmitir. Para poder resolver este problema sin más complicaciones, se avisarán y transmitirán a la impresora de gráficos, sólo &27F bytes de gráfico. Esto significa que la última serie de puntos de la pantalla no aparece en el hardcopy. Esta es una desventaja con la que nos encontraremos en casi todos los casos.

Vayamos ahora directamente al programa en sí.

Para leer los siete bytes que se encuentran uno debajo de otro, se constituirá una tabla, donde se encuentran almacenados los siete bytes actuales.

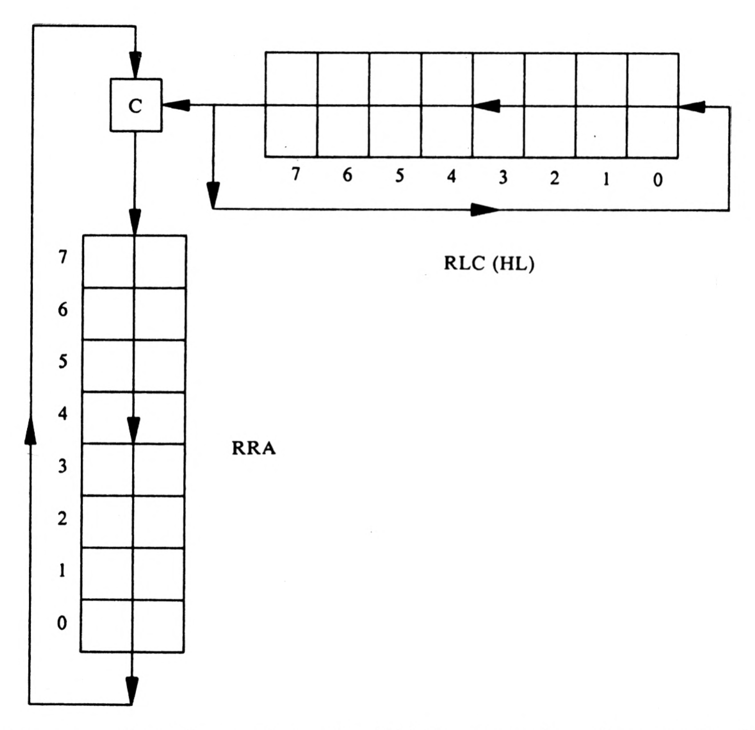
La octava dirección de la tabla contiene siempre la dirección del primero de los siete bytes, que deben tratarse a continuación. Esta tabla se creará de nuevo a partir de la etiqueta NELI1 para las siete siguientes líneas. Para ello se utilizará la rutina del sistema SCR NXT LINE, que eleva la dirección de pantalla entregada en HL de tal forma que se presenta en la siguiente línea de puntos de la pantalla. En el bucle de etiqueta NEBIT se encuentra la transformación de formato horizontal 8 bits en formato vertical 7 bits.

Para ello, empezando por el último elemento de la tabla (TABLET), se cargará la correspondiente dirección de byte en el registro HL. Mediante RLC (HL), se rotará el byte hacia la izquierda, cargando el MSB (bit de valor más elevado) en el carry.

De este modo, el punto de imagen se encuentra almacenado en el carry, y mediante RRA (rotación de ACCU a la derecha) se introduce en el ACCU. Así empieza el bucle desde el pricipio. Después de siete pasadas, el ACCU contiene los siete bits más altos (=puntos) de las líneas de pantalla.

Este bucle es un ejemplo extraordinario para aclarar el rendimiento de las instrucciones del lenguaje máquina dentro de una tarea concreta.

Observe de nuevo el modo de funcionamiento del bucle en base al siguiente esquema:



Después de finalizar el bucle NEBIT, se rota de nuevo el contenido del ACCU. Con ello, nuestra matriz de bits averiguada no activará más el bit Nr. 7, que no puede ser transmitido. Finalmente se reproducirá el byte creado en la impresora.

Este procedimiento se ejecuta para cada byte en total ocho veces. De este modo cada bit rota paso a paso en el ACCU. Existe un efecto interesante para observar este método de programa. Si Vd se fija detalladamente en la pantalla, verá realmente cómo rotan los bytes. Esto se puede ver en todo caso al iniciar una línea, pues el programa debe esperar a la impresora.

Si se han transmitido ya los ocho bits de un byte, se activará mediante el bucle a partir de NEBY1, y con ayuda de la rutina de sistema SCR NEXT BYTE cada elemento de la tabla en el byte siguiente. Durante la transmisión de la actual dirección de pantalla en el registro HL, SCR NEXT LINE incrementa el valor de HL, de modo que direccione el próximo byte en la misma línea.

Mientras no se llegue al final de la línea, y gracias al contador de bytes C, se saltará de nuevo, a la transformación y salida del byte nuevamente direccionado.

Si se llega al final de una línea, se determinará mediante el contador de bytes si deben imprimirse otras líneas. Si es así, se bifurcará hacia la etiqueta NELINE. Allí empieza de nuevo todo el procedimiento. Si se trata de la última línea a imprimir (LD A,E; CP 1), se fijará en 4 la cantidad de bits que han de utilizarse situados uno debajo de otro, y se describirá la tabla a partir de la mitad, sólo con cuatro direcciones de inicio.

Cabe decir que también aquí hemos utilizado un pequeño truco. Después de activar los valores de la tabla con LD DE,TABMIT y LD B,4 para la última línea, debemos pasar naturalmente por encima de la instrucción LD DE,TABANF. Esto se produce normalmente mediante una instrucción JR. Aunque se puede hacer de modo más simple con un byte con valor &21. El ordenador interpreta &21 como opcode de la instrucción LD HL,nn. A ello corresponden los dos bytes siguientes de esta instrucción. El tercer byte es el byte alto de TABANF, esto es &A0. &A0 se encuentra para la instrucción AND B. Las dos instrucciones creadas de esta forma no perjudican a nuestra ejecución del programa, y tampoco cambian ningún contenido de registro importante. Después de estas dos instrucciones, el programa seguirá de nuevo de forma regular con la instrucción LD HL,(TABEND). El efecto deseado, esto es, el salto de la instrución LD DE,TABEND, se conseguirá de este modo.

Aún debemos mencionar dos importantes subprogramas: TABOUT y PRINT.

TABOUT se encuentra para la edición de la tabla. Con él nos referimos a tablas de instrucciones de control en la impresora. La primera secuencia de mando es PRINT. Sitúa el avance de línea en 7/72 de pulgada y provoca un avance de línea. Para comunicar esta secuencia de control a la impresora, se cargará la dirección del primer byte de la lista en HL y se llamará luego TABOUT.

TABOUT reconoce el final de una secuencia en el lugar donde se encuentra el byte nulo.

Aparte de ello, tenemos también las secuencias PRLIIN Y PRREIN. PRLIIN (Printer Line Init) envía la secuencia de control, que traslada a la impresora al submodo de aguja de 8 bits para 639 bytes (es decir, para una línea). PRREIN (Printer RE-init) devuelve la impresora de nuevo al sistema de operación standard.

Para que también pueda adaptar el Hardcopy a otras impresoras, hemos dejado cuatro bytes libres en cada secuencia. Si su impresora no tiene que llevar a cabo ningún avance de línea, introduzca el código 24 en la secuencia PRLIIN en lugar del código 10 (LF).

La rutina Print posibilita la edición impresa. Primero se disminuye el contador PRTCOU, y se salta, en caso de que el contador sea cero. De este modo se enviarán 639 bytes en lugar de 640 por línea. Al final se examinará con MCPRBU si la impresora se encuentra "busy" (inglés.: atareada). Si es así, seguirá el examen, en otro caso, se enviará con MCPRCH el byte, que se encuentra en el ACCU, a la impresora.

```
A000             10                   ; Harcopy de A.R. 18/10/85
A000             20
; para Epson FX80 y compatibles
A000             30
; aplicable a CPC 464,664 y 6128
A000             40                   ; sin cambios
A000             50             ORG   &a000
A000             60   SCGELO EQU  &bc0b ; Screen Get Location
A000             70   SCNELI EQU  &bc26 ; Screen next Line
A000             80   SCNEBY EQU  &bc20 ; Screen next Byte
A000             90   MCPRCH EQU  &bd2b ; MC print Character
A000            100   MCPRBU EQU  &bd2e ; MC Printer Busy ?
A000            110
A000 210000     120             LD    hl,prinit ; inicialización
A003 CD0000     130             CALL  tabout ; de impresora
A006 CD0BBC     140             CALL  scgelo
A009 57         150             LD    d,a
A00A 1E00       160             LD    e,0
A00C 19         170             ADD   hl,de
; dirección de punto superior izquierdo
A00D 220000     180             LD    (tabend),hl
A010 1E1D       190             LD    e,29 ; contador de linea
A012 3E07       200             LD    a,7
A014 320000     210             LD    (bitanz),a
; Bits por linea impresa
A017 D5         220   NELINE PUSH de ;salvar contador de linea
A018 0608       230             LD    b,8
; cambiar 8 direcciones de inicio de linea
A01A 7B         240             LD    a,e ; si no
A01B FE01       250             CP    1 ; última linea
A01D 20FE       260             JR    nz,ok ; no, todo ok
A01F 3E04       270             LD    a,4 ; sino solo 4 Bit por
A021 320000     280             LD    (bitanz),a ; linea impresa
A024 110000     290             LD    de,tabmit;y rellenar tabla con
A029 21         310             DB    &21
; "destruir" linea siguiente
**** Linea  260  : OK=&A02A
```

```
A02A 110000   320  OK     LD   de,tabanf
A02D 2A0000   330         LD   hl,(tabend) ; HL es la nueva
A030 EB       340  NELI1  EX   de,hl
; primera introducción en la tabla
A031 73       350         LD   (hl),e
A032 23       360         INC  hl
A033 72       370         LD   (hl),d
A034 23       380         INC  hl
A035 EB       390         EX   de,hl
A036 CD26BC   400         CALL scneli
; dirección de linea siguiente
A039 10F5     410         DJNZ neli1 ; tomar y almacenar
A03B 21000    420         LD   hl,prliin ; conectar
A03E CD0000   430         CALL tabout ; modo muestra Bit
A041 217F02   440         LD   hl,&27f ; pero enviar
A044 220000   450         LD   (prtcou),hl ; solo 639 Bytes
A047 015008   460         LD   bc,&850 ;c=contador Bytes=80
A04A C%       470  NEBY   PUSH bc
A04B 3A0000   480         LD   a,(bitanz) ; número de
A04E 47       490         LD   b,a ; Bits impresos
A04F 97       500         SUB  a ; borrar acumulador
A050 210000   510         LD   hl,tablet
; con la última tabla
A053 56       520  NEBIT  LD   d,(hl) ; iniciar elemento
A054 2B       530         DEC  hl
A055 5E       540         LD   e,(hl) ; dirección Byte
A056 2B       550         DEC  hl ; leer
A057 EB       560         EX   de,hl
A058 CB06     570         RLC  (hl) ; rotar Byte
A05A 1F       580         RRA  ; rotar Carry en el acumulador
A05B EB       590         EX   de,hl
A05C 10F5     600         DJNZ nebit ; Bit siguiente
A05E 1F       610         RRA  ; no emplear Bit 7
A05F CD0000   620         CALL print ; Edición
A062 C1       630         POP  bc
A063 10E5     640         DJNZ neby ; 8 veces por Byte
A065 210000   650         LD   hl,tabanf ; 7 elemen. de tabla
A068 0607     660         LD   b,7 ; aumentar
A06A 5E       670         LD   e,(hl) ; un Byte
```

```
A06B            680            INC  hl ; hacia derecha
A06C 56         690            LD   d,(hl)
A06D 2B         700            DEC  hl
A06E EB         710            EX   de,hl
A06F CD20BC     720            CALL scneby
A072 EB         730            EX   de,hl
A073 73         740            LD   (hl),e
A074 23         750            INC  hl
A075 72         760            LD   (hl),d
A076 23         770            INC  hl
A077 10F1       780            DJNZ neby1
A079 0608       790            LD   b,8
A07B 0D         800            DEC  c ; aun no es final de linea
A07C 20CC       810            JR   nz,neby ; luego Byte siguiente
A07E D1         820            POP  de
A07F 1D         830            DEC  e ; última linea ?
A080 2095       840            JR   nz,neline ; no,luego siguiente
A082 210000     850            LD   hl,prrein
; reinicializar impresora
A085 18FE       860            JR   tabout ; luego final
A087            870                 ;
**** Linea  130  : TABOUT=&A087
**** Linea  430  : TABOUT=&A087
**** Linea  860  : TABOUT=&A087
A087 7E         880     TABOUT LD   a,(hl) ; tabla en
A088 FE00       890            CP   0; dirección HL
A08A C8         900            RET  z ; hasta
A08B 23         910            INC  hl ; Byte nulo
A08C E5         920            PUSH hl ; edición
A08D CD0000     930            CALL print
A090 E1         940            POP  hl
A091 18F4       950            JR   tabout ; Byte siguiente
A093            960                 ;
**** Linea  620  : PRINT=&A093
**** Linea  930  : PRINT=&A093
A093 2A0000     970     PRINT  LD   hl,(prtcou) ;contador de Bytes
A096 2B         980            DEC ; disminuir
A097 220000     990            LD    (prtcou),hl
; y realmacenar
```

```
A09A 4F       1000          LD   c,a
; almacenar caracter
A09B 7C       1010          LD   a,h ; analizar si
A09C B5       1020          OR   l ; contador de Bytes
A09D C8       1030          RET  z ; es igual a cero
A09E 79       1040          LD   a,c ;no,pues editar acumula.
A09F CD2EBD   1050   WAIT   CALL mcprbu ; Printer Busy ?
A0A2 38FB     1060          JR   c,wait ; si, seguir esperando
A0A4 CD2BBD   1070          CALL mcprch ; emitir caracter
A0A7 C9       1080          RET
A0A8          1090
**** Linea  120  : PRINIT=&A0A8
A0A8 1B       1100   PRINIT DB   27 ; ESC
A0A9 31       1110          DM   "1" ;progr. salto línea a 7/72
A0AA 0D00     1120          DW   &000d ; CR y Byte 0
A0AC 0000     1130          DW   0 ; espacios para
A0AE 0000     1140          DW   0 ; variaciones
A0B0          1150
**** Linea  420  : PRLIIN=&A0B0
A0B0 0D       1160   PRLIIN DB   13
A0B1 18       1170          DB   24
; CANcel; eventualmente sustituir por 10=LF
A0B2 1B       1180          DB   27
A0B3 2A       1190          DM   "*" ; act. modo patrón de Bit
A0B4 01       1200          DB   1 ; modo 1
A0B5 7F02    1210          DW   &027f
A0B7 00      1220          DB   0 ; 0-Byte
A0B8 0000    1230          DW   0 ; espacio para
A0BA 0000    1240          DW   0 ; variaciones
A0BC         1250
**** Linea 850  : PRREIN=&A0BC
A0BC 1B      1260   PRREIN DB   27 ; ESC
A0BD 40      1270          DM   "@" ; normalizar
A0BE 0D00    1280          DW   &000d ; CR y 0-Byte
A0C0 0000    1290          DW   0 ; espacio para
A0c2 0000    1300          DW   0 ; cambios
A0C4         1310
**** Linea 210  : BITANZ=&A0C4
**** Linea 280  : BITANZ=&A0C4
```

```
**** Linea 480  : BITANZ=&A0C4
A0C4 07      1320  BITANZ DB   7
**** Linea 450  : PRTCOU=&A0C5
**** Linea 970  : PRTCOU=&A0C5
**** Linea 990  : PRTCOU=&A0C5
A0C5 7F02    1330  PRTCOU DW   &27f
**** Linea 320  : TABANF=&A0C7
**** Linea 650  : TABANF=&A0C7
A0C7         1340  TABANF DS   6
**** Linea 290  : TABMIT=&A0CD
A0CD         1350  TABMIT DS   7
**** Linea 510  : TABLET=&A0D4
A0D4         1360  TABLET DS   1
**** Linea 180  : TABEND=&A0D5
**** Linea 330  : TABEND=&A0D5
A0D5         1370  TABEND DS   2


Programa : hardcopy[
Inicio : &A000  Fin : &A0D6
Extensión : 00D7
0 Error
Tabla de variables :
SCGELO BC0B  SCNELI BC26  SCNEBY BC20 MCPRCH BD2B
MCPRBU BD2E  NELINE A017  OK     A02A NELI1  A030
NEBY   A04A  NEBIT  A053  NEBY1  A06A TABOUT A087
PRINT  A093  WAIT   A09F  PRINIT A0A8 PRLIIN A0B0
PRREIN A0BC  BITANZ A0C4  PRTCOU A0C5 TABANF A0C7
TABMIT A0CD  TABLET A0DA  TABEND A0D5
```

```
10 REM Hardcopy para todos los Amstrad
20 REM solicitud con call &a000 en MODE 2
30 FOR i=&A000 TO &A0C6
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 19650 THEN PRINT"Error en DATAs":END
70 PRINT"ok!":END
80 DATA 21,A8,A0,CD,87,A0,CD,0B
90 DATA BC,57,1E,00,19,22,D5,A0
100 DATA 1E,1D,3E,07,32,C4,A0,D5
110 DATA 06,08,7B,FE,01,20,0B,3E
120 DATA 04,32,C4,A0,11,CD,A0,06
130 DATA 04,21,11,C7,A0,2A,D5,A0
140 DATA EB,73,23,72,23,EB,CD,26
150 DATA BC,10,F5,21,B0,A0,CD,87
160 DATA A0,21,7F,02,22,C5,A0,01
170 DATA 50,08,C5,3A,C4,A0,47,97
180 DATA 21,D4,A0,56,2B,5E,2B,EB
190 DATA CB,06,1F,EB,10,F5,1F,CD
200 DATA 93,A0,C1,10,E5,21,C7,A0
210 DATA 06,07,5E,23,56,2B,EB,CD
220 DATA 20,BC,EB,73,23,72,23,10
230 DATA F1,06,08,0D,20,CC,D1,1D
240 DATA 20,95,21,BC,A0,18,00,7E
250 DATA FE,00,C8,23,E5,CD,93,A0
260 DATA E1,18,F4,2A,C5,A0,2B,22
270 DATA C5,A0,4F,7C,B5,C8,79,CD
280 DATA 2E,BD,38,FB,CD,2B,BD,C9
290 DATA 1B,31,0D,00,00,00,00,00
300 DATA 0D,18,1B,2A,01,7F,02,00
310 DATA 00,00,00,00,1B,40,0D,00
320 DATA 00,00,00,00,07,7F,02
```

12. El Timing correcto para el CPC

Para que no olvide que mientras Vd programa, el tiempo va transcurriendo, hemos desarrollado un reloj por software. Así, por una parte podrá ahorrarse discusiones con la familia y los amigos, y por otra tendrá un medio más, para mejorar el trabajo con programas de aplicación.

Los programas tales como un reloj de software sólo pueden ser escritos con ayuda del control por interrupciones. La programación de interrupciones es aún más difícil que la programación "normal" en lenguaje máquina. Ya es bastante difícil comprobar los programas que están escritos en lenguaje máquina. En programas que utilizan interrupciones, es prácticamente del todo imposible realizar la comprobación bajo condiciones reales, puesto que este tipo de programas siempre dependen del "Timing".

De todas formas la programación de interrupciones ofrece posibilidades insospechadas. El ordenador no funcionaría de ningún modo sin interrupciones. Una de las tareas básicas de las interrupciones es por ejemplo llamada del teclado. Naturalmente también se desarrollan internamente las instrucciones de interrupciones BASIC sobre la interrupción interna.

Ya podemos empezar. Se llevará el control y la sincronización, mediante el oscilador de cuarzo que se encuentra en todo ordenador, con una secuencia concreta de toda la ejecución. Ese cuarzo es con ello, el reloj interno del ordenador. Muchos ordenadores contienen varios osciladores de este tipo. A través de uno de estos osciladores, y a través de Gate Array, se conectará a intervalos regulares la patilla Interrupt Request (IRQ) del procesador Z80 en a Low en el Amstrad. Con ello se bifurcará a partir del programa en lenguaje máquina a una dirección dependiente del modo interrupción.

El Z80 opera en el Amstrad dentro del modo interrupt 1. Con ello, un IRQ provoca un RST &38 o la instrucción CALL &0038. El IRQ aparece en el ordenador Amstrad 300 veces por segundo.

Si la interrupción no se ha desconectado con DI, la dirección actual de ejecución de programa se sitúa en una pila y se bifurca a la dirección &38. Allí se encuentra nuevamente un salto a la dirección &B989 (6128: &B941/ 664: &B941), esto es, a la RAM permanentemente activada, donde empieza la rutina de interrupción.

A partir de aquí se selecciona la ROM inferior y en caso necesario, se salta a la rutina Service Interrupt en la dirección &00B1. Allí se llevará a cabo p.e. la llamada al teclado o el aumento de la variable BASIC TIME.

Una segunda rutina ROM sirve para el enlace de las interrupciones BASIC. También se llevará a cabo el control general del chip de sonido mediante rutinas de interrupción.

Para enlazar una rutina interrupción propia con la que se está tratando, situamos un patch sobre el salto a la rutina Service Interrupt, que debe llamar a nuestra rutina. Al final de esta rutina debe producirse naturalmente, una bifurcación hacia la verdadera rutina Service Interrupt, o sea, hacia la dirección &00B1.

Después de estas indicaciones podemos desarrollar nuestras propias rutinas sin limitaciones y de forma independiente, y después enlazarlas. A partir del punto en que se produce el enlace, aparecerá la rutina automáticamente cada 1/300 segundos, sin que se influya de modo apreciable la ejecución del programa que se esté llevando a cabo en ese momento.

Esto significa en nuestro ejemplo de software de reloj, que éste se ejecuta de modo directo, sin nuestra intervención. Mientras el ordenador siga conectado, seguirá apareciendo la hora actual en la pantalla, en la posición deseada. Ahora vamos a tratar la verdadera rutina.

Al principio del programa se encuentra la pequeña rutina de inicialización, que desvía la instrucción CALL de la rutina Service Interrupt a nuestra propia rutina. Las instrucciones siguientes modifican la rutina Init de tal modo, que con una nueva llamada vuelva a provocar el desactivado de la rutina.

La rutina empieza a partir de la etiqueta START. Allí se asignará primero la dirección de salto atrás para la rutina Service Interrupt en el Stack.

Esto significa que las propias rutinas se pueden cerrar después simplemente mediante RET, y luego se salta automáticamente a la dirección &00B1.

El primer contador COUNT se disminuirá en cada llamada en uno. Con ello se consigue que sólo una de cada 300 veces, o sea, una vez segundo, aparezca nueva la hora. En este sentido es mejor dejar pasar el menor tiempo posible, sinó el ordenador va cada vez más lento. Si el contador COUNT es igual a 0, se cargará de nuevo para el siguiente procedimiento con 300. Luego empieza el verdadero programa de reloj.

Para almacenar la hora se utilizan 3 bytes. Un byte para las horas, otro para los minutos, y otro para los segundos. Para ello se almacenará la hora actual en bytes de horas, p.e. 16 para las 16 horas. El valor de cada byte se almacenará en el formato BCD. Esto significa que el valor de cada cifra decimal se almacenará en 4 bits. BCD significa decimal codificado en binario. En base a las particularidades del sistema hexadecimal se puede escribir también lo siguiente. El decimal 16 es en el formato BCD &16. El formato BCD es en nuestro caso más efectivo en razón al tiempo utilizado, que el simple almacenamiento del valor. El cálculo con números 1-byte-BCD es apenas un poco más lento que con números normales, pues el Z80 consta de la instrucción DAA, especial para la aritmética BCD. La salida de un número BCD en pantalla, es mucho más simple y rápida en comparación con números almacenados normalmente. En este sentido se pierde el mínimo de tiempo de cálculo.

En las rutinas de interrupción deberíamos colocar un valor especial en el factor de cálculo de tiempo.

Para seguir presentando la hora por segundos, debemos introducir la dirección del byte de segundos en el registro HL, y el número máximo de segundos (o sea, 60) en el subprograma ZEIT.

El subprograma ZEIT incrementa el número BCD, que se encuentra en la dirección dada en uno, y examina si se ha llegado al valor máximo. Si es así, se borrará el Carry-Flag, que en caso contrario será activado.

Si aún no se ha conseguido el valor máximo, después del salto a la rutina principal, se saltará inmediatamente a la emisión de la hora actual. Esto es posible porque los minutos y las horas sólo cambian, cuando han pasado 60 segundos

Si se llega al valor máximo, ZEIT activa el byte en cuestión de nuevo a 0, ya que después de 59 segundos, o minutos, aparece otra vez el 0, igual que después de 24 horas. Cuando se llega al valor máximo de segundos, se carga HL con la dirección de minutos, y, para aumentarlos, se llama de nuevo ZEIT. Al alcanzar también el valor máximo de minutos (60), se aumentan también las horas, cargándose B previamente con 24. Ahora los 3 bytes ya están activados a la hora actual y pueden ser sacados por la pantalla.

Para ello se carga la posición de memoria POS con la posición de pantalla para la salida. Después, con la rutina BCDYOU (BCD BYte OUt), se ofrecen las horas, minutos y segundos separados mediante dos puntos. Con BCBYOU se aprecia la ventaja de los números BCD. El ACCU es cargado con &30, el código ASCII de "0". Después se va rotando cada vez con RLD- para lo que HL debe señalar siempre el byte en cuestión -una cifra BCD en los cuatro bits inferiores del ACCU. Con ello se encuentra en el ACCU inmediatamente el código ASCII de la cifra en cuestión, y puede ser mostrado. La salida se produce a través de la llamada del subprograma PRINT, que en cada salida calcula y almacena la posición para la salida siguiente. Ahora le presentamos el listado assembler completo.

```
A000            10
; reloj de software para todos los Amstrad
A000            20                  ; H.D. 22/9/85
A000            30
A000            40                  ; rutina de inicializaciòn
A000 210000     50          LD   hl,inicio; rutina interrup.
A003 2251B9     60          LD   (&b951),hl ; hacer patch
A006            70                  ; para 464: ld (&b949),hl
A006 21B100     80          LD   hl,&b1
A009 2201A0     90          LD   (&a001),hl
A00C C9         100         RET
A00D            110
A00D            120  WRITE  EQU  &bdd3 ; (464, 664 y 6128 !!)
A00D            130  INTCOU EQU  300
A00D            140  POSITI EQU  &4700 ; modo 2
A00D            150  DOPPUN EQU  &3a
A00D            160
**** Linea  50  : INICIO=&A00D
A00D 21B100     170  INICIO LD   hl,&b1;direc. salto de retorno
A010 E5         180         PUSH hl ; rutina Service-Interrupt
A011 2A0000     190         LD   hl,(count) ; tomar contador
A014 2B         200         DEC  hl
; disminuir, para que solo
A015 220000     210         LD   (count),hl
A018 7C         220         LD   a,h
; se ejecute "el resto" cada segundo
A019 B5         230         OR   l ; esto es :el contador no es
A01A C0         240         RET  nz ; cero, luego listo
A01B 212C01     250         LD   hl,intcou ;inicial. de nuevo
A01E 220000     260         LD   (count),hl ; el contador
A021            270
A021 210000     280         LD   hl,sec ;direc. Bytes por seg.
A024 0660       290         LD   b,&60 ; max. 60 (BCD) segundos
a026 CD0000     300         CALL zeit ; aumentar segundos
A029 38FE       310         JR   c,anzei
; no es aun 60, luego presentaciòn
A02B            320         CALL zeit ; aumentar minutos
A02E 38FE       330         JR   c,anzei
```

```
A030 0624      340          LD   b,&24 ; max. 24 horas
A032 CD0000    350          CALL zeit ;
**** Linea  310  : ANZEI=&A035
**** Linea  330  : ANZEI=&A035
A035 210047    360  ANZEI   LD   hl,positi
; posición para salida
A038 220000    370          LD   (pos),hl
A03B 210000    380          LD   hl,stund
A03E CD0000    390          CALL bcbyou
; Formato-BCD Byte de salida
A041 3E3A      400          LD   a,doppun
A043 CD0000    410          CALL print
A046 CD0000    420          CALL bcbyou ; editar minutos
A049 3E3A      430          LD   a,doppun
A04B CD0000    440          CALL print
A04E CD0000    450          CALL bcbyou ; editar segundos
A051 C9        460          RET
A052           470
A052           480
; subprograma para aumentar el contador de tiempo
**** Linea  300  : ZEIT=&A052
**** Linea  320  : ZEIT=&A052
**** Linea  350  : ZEIT=&A052
A052 7E        490  ZEIT    LD   a,(hl) ; aumentar
A053 C601      500          ADD  a,1 ; tiempo anterior
A055 27        510          DAA   ; Bytes están en formato BCD
A056 77        520          LD   (hl),a ; almacenar
A057 B8        530          CP   b ; alcanzado el máximo
A058 D8        540          RET  c ; no, luego listo
A059 97        550          SUB  a ; si, luego activar
A05A 77        560          LD   (hl),a ; con "0"
A05B 2B        570          DEC  hl
; otra vez minutos (horas)
A05C C9        580          RET
A05D           590
A05D           600
; rutina para salida de un Byte BCD
**** Linea  390  : BCBYOU=&A05D
**** Linea  420  : BCBYOU=&A05D
```

```
**** Linea  450  : BCBYOU=&A05D
a05D 3E30     610  BCBYOU LD   a,&30
; edición de números (asc("0")=&30)
A05F ED6F     620         RLD
; rotar cifra mayor valor en acumulador
A061 CD0000   630         CALL print
A064 ED6F     640         RLD
A066 CD0000   650         CALL print ; cifra menor valor
A069 ED6F     660         RLD   ; reproducir valor
A06B 23       670         INC  hl
; otra vez minutos(segundos)
A06C C9       680         RET
A06D          690
**** Linea  410  : PRINT=&A06D
**** Linea  440  : PRINT=&A06D
**** Linea  630  : PRINT=&A06D
**** Linea  650  : PRINT=&A06D
A06D E5       700  PRINT  PUSH hl
A06E F5       710         PUSH af
A06F 2A0000   720         LD   hl,(pos)
A072 24       730         INC  h
A073 220000   740         LD   (pos),hl
A076 CDD3BD   750         CALL write
A079 F1       760         POP  af
A07A E1       770         POP  hl
A07B C9       780         RET
A07C          790
**** Linea  370  : POS=&A07C
**** Linea  720  : POS=&A07C
**** Linea  740  : POS=&A07C
A07C          800  POS    DS   2
**** Linea  190  : COUNT=&A07E
**** Linea  210  : COUNT=&A07E
**** Linea  260  : COUNT=&A07E
A07E 2C01     810  COUNT  DW   300
**** Linea  380  : STUND=&A080
A080 00       820  STUND  DB   0
A081 00       830         DB   0
**** Linea  280  : SEC=&A082
```

```
A082 00         840   SEC     DB    0
Programa : reloj
Inicio : &A000    Final : &A082
Extensión : 0083
 0 Error
Tabla de variables
WRITE   BDD3   INTCOU 012C   POSITI 4700   DOPPUN 003A
START   A00D   ANZEI  A035   ZEIT   A052   BCBYOU A05D
PRINT   A06D   POS    A07C   COUNT  A07E   STUND  A080
SEC     A082
```

```
10 REM BASIC cargador para hora
20 REM Inicilizacion con Call &a000
30 FOR i=&A000 TO &A07F
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 14784 THEN PRINT"Error en Datas":END
70 'para 464:IF s<>14776 THEN.......
80 PRINT"ok!":END
90 DATA 21,0D,A0,22,51,B9,21,B1
100 '464:..,..,..,..,49,..,..,..
110 DATA 00,22,01,A0,C9,21,B1,00
120 DATA E5,2A,7E,A0,2B,22,7E,A0
130 DATA 7C,B5,C0,21,2C,01,22,7E
140 DATA A0,21,82,A0,06,60,CD,52
150 DATA A0,38,0A,CD,52,A0,38,05
160 DATA 06,24,CD,52,A0,21,00,47
170 DATA 22,7C,A0,21,80,A0,CD,5D
180 DATA A0,3E,3A,CD,6D,A0,CD,5D
190 DATA A0,3E,3A,CD,6D,A0,CD,5D
200 DATA A0,C9,7E,C6,01,27,77,B8
210 DATA D8,97,77,2B,C9,3E,30,ED
220 DATA 6F,CD,6D,A0,ED,6F,CD,6D
230 DATA A0,ED,6F,23,C9,E5,F5,2A
240 DATA 7C,A0,24,22,7C,A0,CD,D3
250 DATA BD,F1,E1,C9,00,00,2C,01
```

En el caso de que no posea ningún Assembler, hemos confeccionado también un cargador BASIC.

Ahora presentamos un pequeño programa BASIC, con el que puede poner el reloj en hora de modo más cómodo. Si desea desconectar totalmente todo el reloj, esto es, desactivar la rutina de interrupción, debe llamar la rutina por segunda vez con CALL &A000. Una nueva puesta en hora podrá efectuarse solamente tras la entrada de

```
POKE &A001,&D:POKE &A002,&A0
```

y llamando con CALL.

```
10 REM Graduar reloj
20 MEMORY &9FFF
30 MODE 2
40 ' LOAD"uhr1.obj
50 LOCATE 15,7:PRINT"G r a d u a r   r e l o j"
60 LOCATE 1,11
70 INPUT"Indicacion de 12 o 24 horas (12/24) ?";e
80 PRINT
90 IF e<>12 AND e<>24 THEN 70
100 POKE &A031,VAL("&"+STR$(e))
110 zeibas=&A080
120 INPUT"Hora (hh,mm,ss)";h,m,s
130 PRINT
140 POKE zeibas, VAL("&"+STR$(h)):POKE zeibas+1,VAL("&"+STR$
(m)):POKE zeibas+2,VAL("&"+STR$(s))
150 INPUT "Posicion (columna,linea)";sp,ze
160 IF sp<1 OR ze<1 THEN 150
170 IF sp=1 THEN sp=257
180 POKE &A036,ze-1:POKE &A037,sp-2
190 ' CALL &A000
```

PARTE 3

Consejos & Trucos para el lenguaje máquina

13. Programar en lenguaje máquina

Cuando la velocidad es uno de los factores más importantes para la programación, el programador está obligado, directa o indirectamente a servirse del al lenguaje máquina. Directamente significa que el cógigo máquina debe introducirse en el ordenador. Para ello existen un gran número de programas assembler, que facilitan este trabajo.
En este apartado llamaremos forma indirecta a la utilización de un compilador. Si se dispone p.e. de un compilador BASIC, el programa de realización propia puede escribirse como de costumbre en el lenguaje de programación BASIC. El compilador, debe traducir luego este BASIC a lenguaje máquina, el cual se tratará y almacenará separado del código fuente (BASIC).

Para tener una idea de la esencia del lenguaje máquina, vamos a crear algunas pequeñas rutinas mediante ensamblaje directo, y presentaremos sus ventajas en cuestión de tiempo, las cuales están por encima de un lenguaje interpretador, como es el BASIC en el CPC 464/664. Para ello no necesitamos ningún programa de assembler. Para llevar el código máquina a la memoria de trabajo, nos serviremos de pequeños programas cargadores BASIC.

Iniciemos el siguiente pequeño programa BASIC.

```
50 MODE 2
100 FOR DIRECCION=49152 TO 65535
110 POKE DIRECCION,255
120 NEXT
```

Descripción del programa:

Los 16K RAM superiores de la memoria CPC (&C000-&FFFF) están reservados para la pantalla. Los contenidos numéricos de las posiciones de memoria representan la trama de puntos, que llevados a pantalla, producen caracteres con pleno sentido.

Con nuestro programa de ejemplo introducimos todos los puntos en pantalla.

50: La instrucción MODE 2 borra la pantalla y activa el offset en la esquina superior izquierda.

100: El bucle FOR-NEXT empieza con la primera dirección de pantalla y finaliza con la última.

110 El valor 255 se POKEa en todas las posiciones de memoria.

La ejecución de este programa necesita aprox. 40 segundos. Con 16384 posiciones de memoria para asignar, significa que cada asignación de una posición de memoria dura aprox. 2,4 milisegundos.
Para conseguir una comparación de tiempo con un programa en lenguaje máquina, que esté constituído más o menos como el programa BASIC, ejecute el siguiente programa.

```
10 MODE 2:SUMA=0
20 FOR I=40960 TO 40979
30   READ a$:VALOR=VAL("&"+a$)
40   POKE DIRECCION,VALOR:SUMA=SUMA+VALOR:NEXT
50 IF SUMA<>1531 THEN PRINT "Error en Datas":END
60 PRINT "PULSACION DE TECLA INICIA RUTINA MAQUINA"
70 CALL &BB06:CALL 40960
80 END
90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

Con esta rutina assembler se puede reducir el tiempo de ejecución en aprox. 0,2 segundos, esto es, 12 milisegundos por posición de memoria. Aunque esta rutina no representa tampoco la posibilidad más rápida, esta comparación debería aclarar las posibilidades del ensamblado.

Indicaciones para el programa:

El programa en lenguaje máquina consta de valores hexadecimales de las líneas DATA 90 y 100. En el bucle FOR-NEXT (línea 20-40) se leen estos valores transformados en un valor numérico (LINEA 30) y luego se POKEa en la zona de memoria 40960-40979. La instrucción CALL &BB06 en la línea 70 llama una rutina del sistema operativo, que activa el teclado tanto tiempo como sea necesario hasta que se pulse una tecla cualquiera. Luego se lleva a cabo la primera instrucción. Ahora sigue CALL 40960, que llama a la rutina en lenguaje máquina, escrita en memoria por el bucle FOR-NEXT.
Al finalizar, el ordenador vuelve a ponerse bajo la custodia del interpretador BASIC y lo comunica con READY, después de lo cual, se llega a la línea 80 (END).

¿Qué significan pues los valores hexadecimales de las líneas DATA?

Para poder dar una respuesta satisfactoria, debemos observar primero la CPU del Z80, que es el corazón del ordenador Amstrad.

Después de la introducción en el lenguaje de la CPU (Consejos y trucos/ Tomo 1) estudiaremos ahora de forma más profunda esta caja de trucos que es el ordenador. Vamos a presentar instrucciones importantes, potentes y rápidas del Z80, y las ilustraremos mediante rutinas prácticas como ejemplo.

Para comprenderlas debemos tener un buen conocimiento de los registros, cómo se han constituído, su funcionamiento, y la eficiencia de su aplicación.

13.1 Los registros del Z80

Los registros son posiciones de memoria dentro de la CPU. Aunque el Z80 es un procesador de 8 bits, dispone también de registros de 16 bits.

Los símbolos de los registros son los siguientes:

Registros de 8 bits: A, B, C, D, E, F, H, L
Registros de 16 bits: BC, DE, HL, SP, PC

Aquí se encuentran sólo los registros más importantes. Los restantes no los vamos a nombrar en este apartado, pues no son necesarios para la comprensión básica. También deseo remitir a los siguientes apartados de este capítulo, a aquel lector que ya se haya dado cuenta de que algunos registros de 16 bits se pueden combinar con registros de 8 bits. Esto tiene un razonamiento lógico, pero de ello hablaremos más adelante.

13.1.1 Registros de 8 bits

No todos los registros de 8 bits tienen la misma importancia. El registro A y el registro F toman una posición muy especial por encima de los demás, de modo que la enumeración debería ser de este modo:

A,F, B,C,D,E,H,L

El registro A (Acumulador)

Todas las operaciones lógicas y aritméticas de 8 bits se realizan a través del registro A. Los resultados de estas operaciones se encuentran disponibles en el ACCU, mientras no se cambie el otro registro.

1. Ejemplo de una adición de dos números:

```
LD  A,6     Carga ACCU con 6
LD  D,3     Carga reg. B con 3
ADD A,D     Suma contenido de D al ACCU
```

Después de la operación ADD A,D en A se encuentra el valor 9, y en D se mantiene el valor 6.

2. Ejemplo de una substracción de dos números:

```
LD   A,&54    Carga ACCU con &54
LD   L,&54    Carga reg.L con &54
SUB  A,L      Resta contenido de L del ACCU
```

Después de esta substracción, en el acumulador se encuentra el valor 0.

Estas operaciones aritméticas descritas no influyen sólo en el contenido del ACCU. Otra influencia se produce también en el registro flag. Esto explica distintas repercusiones, que pueden aparecer en las operaciones con el Accu. Para representar estas repercusiones, se activan de nuevo los bits especiales y concretos del registro F.
En base a estas informaciones que el registro flag ofrece, se pueden llevar a cabo operaciones condicionales. Vayamos ahora a la descripción de este registro.

El registro F (registro flag)

Como ya habría reconocido, a través del contenido del registro F, puede obtenerse información sobre los resultados de operaciones lógicas y aritméticas.
El flag Zero es sólo uno de los seis flags que dispone este registro, aunque junto con el bit-carry (C) bit de acarreo es el más importante.

La estructura del registro Flag:

Posición de los flags dentro del registro flag.

Símbolo	S	Z		H		P/V	N	C
Bit Núm.	7	6	5	4	3	2	1	0

Descripción del flag:

S:	Sign (Signo)	0 Resultado positivo 1 Resultado negativo
Z:	Zero (Cero)	0 Resultado distinto a cero 1 Resultado igual a cero
H:	Halfcarry	0 No hay acarreo de bit 3 a bit 4 en el ACCU 1 Acarrero de bit 3 a bit 4
P/V:	Parity/Overflow	0 Paridad impar / No hay acarreo de bit 6 a 7 1 Paridad par / Acarreo de bit 6 a 7 (otras funciones después)
N:	Substracción flag	0 Después de ejecución de una adición 1 Después de ejecución de una substracción
C:	Carry	0 No hay acarreo de bit 7 a 8 1 Acarreo de bit 7 a 8

El Bit Sign (S)

Con los 8 bits de un byte se pueden representar 2^8=256, esto es de 0 a 255.

Podemos representar números con signo de -128 a 127. El valor del número se representa mediante los bits 0 a 6 y el signo por el bit 7. Si el signo es positivo S toma el valor 0. Si el signo es negativo S se activa a 1

El número binario 1111111 representa al número decimal -1.

El bit Zero (Z)

Si se altera un byte mediante una operación lógica o aritmética o a través de una instrucción de rotación o de desplazamiento, el flag Z se presenta con un 1, de forma que todo byte se transforma en cero. El correspondiente al flag Z es el cero, cuando el valor del byte es distinto a cero.
En una comparación entre dos bytes -p.e. CP A,E - el bit Z será cero, si los registros contienen el mismo valor, y uno, si difieren los contenidos. Esto es de fácil comprensión, si partimos de que la instrucción CP A,s substrae el operador s del ACCU.
De las otras tres funciones del flag Z indicaré la aplicación con la instrucción test BIT b,r. La instrucción BIT 5,D analiza el quinto bit del registro D dentro de su contenido. Si el contenido es uno el flag Z es cero, si el contenido es cero, el flag es uno.

El Bit Half-Carry (H)

El flag H presenta un acarreo del nibble inferior (Bit 0-3) al nibble superior (Bit 4-7) de un byte. El ámbito de aplicación principal es en este caso la aritmética BCD y el ajuste decimal (DAA). Si el nibble inferior contiene un valor superior a 9, debe producirse un acarreo al nibble superior, pues para la representación de un número decimal, un nibble no puede ser superior a 9. Este acarreo se presenta mediante el flag H.

La paridad/ Bit-Overflow (P/V)

En este flag se encuentran distintas funciones.

a) A pesar de la alta calidad de stándar del Hardware actual, es difícil reducir el número de errores que aparecen en la transmisión de datos. La paridad se utiliza normalmente en la transmisión de caracteres en el formato ASCII (7-Bit), donde el bit de paridad se coloca como octavo bit. Si el número de bits de datos en 1 es par, el bit de paridad se sitúa en 1. Si el número es impar, el bit de paridad se sitúa en 0. Si en una transmisión de datos se transmite sólo un bit falso, la paridad ya no coincidirá con el bit de paridad. Cuando esto se advierte, debe llevarse a cabo una nueva transmisión de ese byte.

b) Si en una adición o substracción el resultado es mayor que +127, se influye erróneamente el bit 7, que está reservado para el signo. Esto se presenta mediante el flag P/V.

c) En una transmisión de bloque o una instrucción de busca de bloque, este flag se influenciará en función del registro BC. Este registro tiene en la instrucción mencionada, la función de contador. Si el registro de números BC es igual a 0, el flag P/V se activa en 0. Si BC es distinto a 0, P/V será igual a 1.

d) En las instrucciones LD A,I y LD A,R el flag P/V contiene el valor de Interrupt-Enable-Flip-Flops IFF2. Con ello es posible leer y/o almacenar el contenido del IFF2.

El bit-substracción (N)

El programador, siguiendo la costumbre no llevará a cabo ninguna aplicación de este flag. La aplicación verdadera de este flag se encuentra en el ajuste decimal (DAA) después de una adición o una substracción. Este ajuste se lleva a cabo de modo distinto si se trata de una adición, si se trata de una substracción. El CPU reconoce su reacción ante el DAA a partir del contenido del flag N.

El bit Carry (C)

El bit de acarreo muestra en una adición o substracción, si existe acarreo en esas operaciones.
En las instrucciones de rotación o desplazamiento se utilizará el bit carry como bit noveno.
El carry se activará de nuevo con las operaciones lógicas AND, OR y XOR. Las instrucciones del Z80 se utilizan frecuentemente, en el caso de que deba borrarse el carry.

¿Por qué estas operaciones lógicas borran el flag C?

Para aclararlo, en el siguiente ejemplo trataremos este tema con el valor &D3 en el ACCU:

```
        11010011
AND     11010011
        --------
        11010011
```

Como en esta instrucción, igual que en OR o XOR, no puede aparecer nunca un acarreo, ni en el caso de que se compare el ACCU con otro registro, el carry se activa constantemente en 0. Estas instrucciones resultan ser instrucciones de borrar carry reglamentarias.

Algo sobre las complicaciones en caso de activar, reactivar y utilizar los flags

Puede ser que a primera vista resulte difícil comprender que, p.e. el flag Zero sea exactamente 1, si el registro analizado mediante una operación es 0, y a la inversa.
Esta circunstancia es de fácil comprensión, si se trata el contenido del flag como valor real.
Un flag puede tomar dos valores reales, verdadero y falso. El 1 será el verdadero y el 0 el falso.
El Z80 dispone de diversas instrucciones de salto condicional. La CPU reconoce a partir de los estados del flag si se cumple la condición que da lugar al salto.
Como ejemplo de la instrucción de salto JP C, dirección,frecuentemente utilizado, explicaremos la relación con el registro-flag. En la adición del registro A con otro valor, aparecería un resultado, cuyo valor sería mayor que &FF. Como ya sabemos, el resultado no ha sido destruido por el desbordamiento del ACCU, pues en este caso el bit carry debe tomarse como bit noveno con el valor 2^8 (=256).
Si se da el caso, el programa debe bifurcarse en un lugar concreto. Esto lo permite la ya mencionada instrucción de salto condicional : JP C, dirección, que se activa si el bit C está activo, o sea, si el bit C es 1.

Si deseamos realizar un salto, cuando el bit C es 0, utilizaremos la instrucción JP NC, dirección que es activa cuando el carry es cero.

Los registros B, C, D, E, H, L

Estos registros desempeñan las mismas funciones. Se les puede asignar directamente un valor, se pueden cargar con valores de otros registros, y se pueden cargar con ellos mismos, lo que es de escasa utilidad, si no se desea provocar un retardo temporal.
Los registros se pueden cargar con el contenido de la RAM y realmacenar de nuevo en ella.

Se pueden operar los registros con el contenido del ACCU de forma lógica y aritmética y/o alterar con instrucciones de rotación y desplazamiento.
Con ello hemos expuesto ya todos los ámbitos de aplicación de estos registros como registros de 8 bits.

13.1.2 Los registros de 16 bits BC, DE, HL, PC y SP

Los registros de 8 bits B y C, D y E y H y L pueden juntarse, de modo que nos encontremos en situación de disponer de registros de 16 bits con instrucciones especiales en una CPU de 8 bits.

Esto es naturalmente un caso especial. ¿Cómo se debe direccionar por ejemplo la posición de memoria 40000 de los 65536 espacios de memoria que tenemos a nuestra disposición? Con una palabra de 8 bits ya se pueden representar aproximadamente 2^8=256 valores o direcciones distintas. Pero si disponemos de dos bytes para un registro de 16 bits, tenemos la posibilidad de 2^{16}=65536 posiciones de memoria.

Los tres registros de 16 bits tienen funciones muy diferenciadas. Aunque la estructura sea la misma, el procesador exige los parámetros necesarios para ciertas operaciones en uno u otro registro.

De este modo se puede tratar al HL como acumulador (ACCU) de 16 bits. A partir de HL con DE o BC se pueden producir tres operaciones aritméticas distintas.

El registro BC se utiliza frecuentemente como contador. Esto se efectúa en las llamadas instrucciones de transferencia de bloques y de búsqueda, que más adelante trataremos con detalle.

El registro PC (Program Counter)

Para que no haya equivocaciones: El registro PC es un 16-Bits puro. La influencia sobre el mismo es limitada.

La serie de instrucciones de un programa se encuentra en una zona de la memoria fuera del procesador. En el mismo procesador está sólo la instrucción actual a ejecutar. El mismo CPU toma esta instrucción de la posición de memoria correspondiente, y cuya dirección se encuentra en el PC. Se puede tratar pues al contador del programa como a un puntero. Este señala constantemente la posición de memoria, donde se encuentra la siguiente instrucción o valor de datos.
Después de que el procesador haya tomado una instrucción de la memoria, el PC volverá de nuevo al estado actual. El PC se incrementa en el valor correspondiente según la longitud de la entrada (existen instrucciones de 1, 2, 3 y 4 byte).

El registro SP (Stack Pointer)

El stack pointer (puntero a la pila)tiene en todo caso una función de indicador. El contenido del SP es la dirección del elemento superior del stack (pila).

¿Qué es pues un stack?

Mientras normalmente se puede acceder en general a tadas las posiciones de memoria, durante el tratamiento de la pila, sólo es posible llegar hasta el elemento más superior de la misma. Esto se observa fácilmente en el ejemplo de una pila de libros. Si se ha llenado una caja de cartón con más de un libro, al principio podremos acceder sólo al libro que se encuentra encima del montón. Si se desea tomar el último, deberemos empezar por sacar los que se encuentran encima.
A este tipo de acceso se le llama principio LIFO. LIFO significa Last-In-First-Out, donde el último elemento introducido, se toma como primer elemento.

El ejemplo con la pila de libros tiene sólo un fallo. El stack (pila) ha sido construído de arriba a abajo. La primera introducción se encuentra en el lugar superior de la pila, las demás pues, estan situadas en posiciones de memoria con números de dirección menores. Esto no tiene demasiada importancia, pues la administración viene dada desde la CPU. De todos modos esto debe saberse en cualquier caso, y sobretodo si se desea colocar el stack en otra posición.

Esto no sucede muy a menudo, pues siempre existe una zona de memoria en la RAM del ordenador destinada al stack. En ningún caso pueden escribirse otros datos encima de éste. Esto llevaría a bloquear el sistema.

A causa de esta independencia, deseo indicar que el SP señala siempre el elemento más superior, esto es, el último elemento introducido en la pila.

De este modo quedan descritos los registros más importantes de la CPU del Z80.
Existen aún otros registros, que no vamos a nombrar aquí, pues no son imprescindibles para estas indicaciones generales.

Después de conocer a grandes rasgos el Z80, nos ocuparemos de las instrucciones de este microprocesador.
Sería una locura intentar presentarlas todas -pues existen unas 500-, y por ello, nos dedicaremos sólo a las que puedan sernos útiles para la resolución de algunos problemas.

13.3 Un ejemplo detallado de la programación en lenguaje máquina

Como primer programa de ejemplo presentaré el borrado de pantalla llevado a cabo al principio del capítulo.

El programa assembler propiamente dicho consta como ya sabemos de la serie de datos de las líneas DATA 90 y 100.

```
90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

El programa BASIC sirvió ya para POKEar estos valores en la memoria a partir de la posición 40960 (&A000).

Estas líneas DATA tienen una significación directa para los Profis-Assembler. Observemos primero detalladamente el programa. A este tipo de escritura se le llama forma mnemotécnica de los listados assembler.
Mnemónicos (no se trata de ningún error de imprenta) es el nombre para las abreviaturas con fuerza de expresión, que se utilizan en este caso.
Si se lee el código Hex 3E FF, se debe disponer de una buena inteligencia y experiencia, para poder reconocer lo que detrás de ello se esconde. La representación mnemotécnica de la misma instrucción es LD A,&FF. A patir de ésta es más fácil reconocer el significado.

```
Carga(LD) ACCU   con el núm-hex FF
LD         A      ,          &   FF
```

El siguiente listado de programa es muy explicativo. En la primera columna se encuentra el número de la posición de memoria, en la segunda, el código hexadecimal y en la tercera el código mnemónico del assembler.

Este listado se complementa con la columna de "comentarios", donde se puede explicar el significado de las instrucciones.

Direción	Código-Hex	Mnemónico		Observación
A000	21 00 40	LD	HL,&4000	cargar contador
A003	01 01 00	LD	BC,&0001	cargar sustraendo
A006	11 00 C0	LD	DE,&C000	primer esp. pantalla
A009	3E FF	LD	A,&FF	cargar acum. con FF
A00B	12	LD	(DE),A	asigna A a direc. DE
A00C	13	INC	DE	aumenta DE en 1
A00D	ED 42	SBC	HL,BC	sustraer BC de HL
A00F	C2 0B A0	JP	NZ,&A00B	Jump hacia &A00B si Flag Zero=0
A012	C9	RET		vuelve pro. principal
A013	00	NOP		no operación

Las direcciones de la columna de la izquierda se refieren siempre al primer byte de cada línea. Por ello, después de las instrucciones presentadas de 3 bytes se saltará siempre dos direcciones. A ello se acostumbrará rápidamente.

Las primeras cuatro instrucciones del listado son conocidas. Los distintos registros se cargan únicamente con valores, que puedan entenderse en seguida.

La primera instrucción nueva de este listado es la instrucción LD(DE),A.
Si en este caso, encontramos el nombre de un registro entre paréntesis, debemos entender el contenido de este registro como dirección. DE contiene en ese momento el valor &C000 (ver tercera línea). Con esta instrucción LD (DE),A se lleva el contenido de A a la posición de memoria &C000.

La segunda instrucción nueva es INC DE. INC, que significa INCrementar. Esta función incrementa el contenido del registro dado, en 1. Después de esta instrucción queda el valor &C001 en el registro DE.

Por lo demás, todos los registros mencionados hasta ahora pueden ser INCrementados.

La instrucción siguiente, también es SBC HL,BC. SBC significa SuBstraer con carry. Al registro HL se le resta el contenido de BC.
Para evitar dificultades con esta instrucción, se debe poner siempre atención que el carry bit activado eventualmente es bueno para provocar sorpresas en esa operación. En caso de dudar del estado del bit C, debería efectuarse antes siempre un borrado del bit de acarreo. Esto se puede llevar a cabo con las instrucciones AND A, XOR A y OR A.
En nuestro programa de ejemplo he renunciado a ello por razones de simplicidad.

La última nueva instrucción de este programa (RETurn y NOP la aprendió en el volumen I de Consejos y Trucos) es un salto condicional, JP NZ,&A00B.
Este se efectuará en el caso de que se cumpla la condición que está unida al salto. En la instrucción de salto con JP NZ, dirección se encuentra la condición NZ, Non Zero o en español, distinto de cero.
Si el flag Zero es 0, indicando con ello que el resultado de la operación anterior es distinto a 0, se efectuará la instrucción de salto.
En nuestro ejemplo ocurre esto, hasta que el registro HL es 0, esto es &4000 (16384) veces. Ello corresponde al número de posiciones de memoria, que están reservados para la zona de pantalla. Si la instrucción SBC HL,BC lleva el registro HL a cero, se activa al mismo tiempo el flag zero. La condición para la siguiente instrucción RET se verifica y el ordenador vuelve al interpretador BASIC. Este se presenta con READY.

Observaciones de rutina en cuanto a la rapidez

Si se elige el lenguaje máquina para resolver un problema de programación, para conseguir una velocidad mayor en el tratamiento, se debe poner también atención a la hora de elegir las instrucciones y la estructura del programa.
En este sentido la rutina presentada no parece la más optima. Esto se demuestra con el pensamiento del lector observador, pues la solución directa se presenta como elegante, aunque en muchos casos depende de las circunstancias y aplicaciones que se deseen.
En la programación assembler no funcionan necesariamente las convenciones, que hacen que la programación sea más eficiente en lenguajes superiores. De este modo, el borrado de una zona de memoria coherente o el de la pantalla se efectúa más rápido con un programa muy largo, que con un programa largo con pocas instrucciones y de aparencia más eficiente. Como en todos los casos, aquí existe también un compromiso entre la velocidad y el esfuerzo de programación del metodo a utilizar.

Para experimentar la velocidad de una rutina no es necesario que coloquemos un reloj al lado del ordenador. Una calculadora de bolsillo trabaja de modo más exacto.
Esto funciona sólo cuando se sabe cuanto tiempo necesita la CPU para su trabajo. Como ejemplo de nuestra rutina, voy a dar ahora unas indicaciones previas. Los números detrás del mnemónico son las indicaciones sobre la duración de las instrucciones en cuestión, en microsegundos.

```
        LD     HL,&4000    5
        LD     BC,&0001    5
        LD     DE,&C000    5
        LD     A,&FF      3.5
------------------------------
sigue   LD     (DE),A     6.5
        INC    DE          3
        SBC    HL,BC      7.5
        JP     NZ,sigue    5
------------------------------
        RET                5
```

Las primeras cuatro líneas del programa y la última, se ejecutan sólo una vez en todo el programa, lo que significa que el tiempo correspondiente se calcula sólo una vez. La suma es $23.5*10^{-6}$ seg.

La parte del programa, que se encuentra marcada con la marca de salto (etiqueta) "sigue", y que he marcado con un subrayado para que sea más claro, es un bucle con fin, que se ejecuta &4000 veces. Luego se multiplica la suma de estas cuatro líneas por &4000. Como cada suma da 22 como resutado, tenemos $\&4000*22*10^{-6}=0.360448$ segundos para la ejecución total del bucle.
Junto con una ejecución de las cinco líneas restantes tenemos un tiempo de ejecución de todo el programa de 0.3604715 segundos.

En resumen, diremos que la instrucción de salto trabaja un poco mas rápido si la condición es negativa, aunque esto no influye en el cálculo general, ya que se trata de pocos microsegundos.

Sigamos con la rutina. Para ello debemos tener claro lo que ocurre cuando se incrementa el contenido de un registro durante la ejecución. En algún momento se alcanza la cifra más alta que el registro puede representar. A partir de entonces, el registro vuelve de nuevo a cero y se puede empezar a contar de nuevo.

Con ello llegamos al punto, donde nuestras aclaraciones deben referirse a explicar la rutina de borrado. Si el registro en cuestión se encuentra en cero, significa que se ha dejado también una posición de memoria que nos interesa. Y por tanto esta es la situación que pone fin a nuestra rutina.

Hasta ahora hemos utilizado un registro de números separado (HL=&4000) junto al registro de direcciones DE. El tiempo suplementario que hemos empleado en la substracción de HL, deseamos ahorrarlo en el siguiente ejemplo. El registro de direcciones DE vuelve de nuevo a cero después de la última dirección &FFFF al INCrementar.
Después de optimizar el contador, nuestro programa aparece de la siguiente forma.

```
        LD    DE,&C000    Dirección de inicio
        LD    A,&FF       Valor a almacenar
sigue   LD    (DE),A      Almacenar
        INC   DE          Renovar direcc.
        JP    NZ,sigue    Salto a condición
        RET               De nuevo al programa principal
```

Esto parece ser una solución práctica. Se han ahorrado tres líneas, dos registros dobles, y, lo que es más importante, la línea SBC HL,BC ya no se ejecuta. Esto ofrecería un ahorro de tiempo de 0.12288 para la serie, lo que corresponde a un 35% con la primera realización del programa.

Desgraciadamente los principiantes de assembler encontrarán aquí, antes o después un problema.

La instrucción INC DE no influye en el registro de flags, de donde la instrucción de salto condicional saca las informaciones. También en el caso de que DE se incremente hasta cero, el flag zero no es influenciado, igual que los restantes flags.

De ello se deduce que la condición de interrupción nunca llega a realizarse:
Primero se asigna &FF a toda la zona de pantalla. Luego se va incrementando el registro de dirección DE, hasta cero. Con ello, se va de nuevo a la posición de memoria más inferior del CPC. Allí se encuentran algunas rutinas, muy importantes para el sistema, que se sobreescriben otra vez. Con lo que inevitablemente se bloquea el ordenador.

De todas formas nuestro trabajo anterior no ha sido del todo en vano. Sólo se debe conseguir de algún modo que después de INC DE, el flag zero se active conforme al resultado de la instrucción INC. La instrucción BIT 7,D sirve para ello. Su descripción continua después del siguiente listado del programa mejorado y ejecutable.

```
        LD   DE,&C000
        LD   A,&FF
sigue   LD   (DE),A
        INC  DE
        BIT  7,D
        JP   NZ,sigue
        RET
```

El registro de dirección DE se carga al principio de la rutina con &C000. A partir de su representacion binaria se reconoce el estado de los 16 bits de esta cifra:

Registro	D		E	
hexad.	C	0	0	0
binar.	1100	0000	0000	0000
Bit Núm.	7654	3210	7654	3210

Como se puede observar, los bits 6 y 7 del registro D se encuentran activados desde el principio, mientras los demás 14 bits son 0. Al incremetar DE, cambian únicamente los 14 bits de menor valor, que al principio son todos igual a 0. Cuando DE tiene el valor &FFFF (11111111 11111111 binario) y se incrementa otra vez, se pasa al desbordamiento.
Las 16 posiciones del registro DE se transforman luego en 0, igual que el bit 6 y 7 de D.

En este punto la instrucción BIT 7,D "espera", influyendo sobre el flag zero, contrariamente a las intrucciones de incremento de 16 bits.
La instrucción activa al flag Z en 1, si el bit analizado se transforma en cero.
En este caso especial se podría utilizar también la instrucción BIT 6,D, pues el bit 6 del registro D se encuentra activado desde el principio, y se anulará al mismo tiempo que el bit 7.

Pero esto sólo lo mencionamos para completar las explicaciones, ya que las dos instrucciones son absolutamente equivalentes.

Al comparar el tiempo de ejecución de esta rutina con el de la primera analizada por nosotros, el resultado es algo más favorable.

```
        LD    DE,&C000    5
        LD    A,&FF     3.5
sigue   LD    (DE),A    6.5
        INC   DE          3
        BIT   7,D         4
        JP    NZ,sigue    5
        RET               5
```

En la comparación de tiempo de este tipo de bucles del programa, podemos pasar por alto las pocas instrucciones, que no pertenecen al bucle, pues no representan una demora importante para nosotros, utilizándose sólo una vez en cada ejecución del programa.
El bucle del último programa utiliza 18.5 microsegundos, mientras el primero utliza 22.
Esto es una clara mejora, pero todavía no suficiente.

En las rutinas descritas hasta este momento, los bucles se ejecutan siempre &4000 (16384) veces.

En cada ejecución, el contenido del ACCU, es decir 1 solo byte en cada pasada, se deposita en una posición de memoria de la pantalla. Esto se efectúa con la instrucción LD(DE),A. Desgraciadamente en el Z80 no existe ninguna instrucción como: LD (DE),HL, pues con ella se podrían asignar dos posiciones de memoria directamente de una sola vez. Esto traería consigo importantes ventajas.
También existen instrucciones que pueden almacenar el contenido de un registro de 16 bits, aunque en ellas no es posible un direccionamiento tan elegante como con la instrucción LD(DE),A, donde sólo se debe INCrementar el registro DE.

Para poder asignar dos bytes con una sola instrucción deberemos servirnos de un truco.
Este truco consiste en abusar de una función de la pila.
Normalmente el stack (pila) se utiliza para p.e. almacenar contenidos de registros momentáneamente, cuando se debe saltar a otra rutina, donde estos registros se utilizan con otros valores. Por ello, el stack se encuentra en un espacio de la RAM, protegido de una intervención por el interpretador BASIC.
El truco consiste en colocar el stack en el espacio de pantalla, para poder utilizar luego la rápida instrucción de pila PUSH rr. Esta instrucción es 6.5 microsegundos más rápida que un LD (DE),A ejecutado dos veces con 7 microsegundos, pero que contiene la actualización del contador de direcciones, que falta aún en la instrucción de carga. Por ello la instrucción PUSH con 6.5 microsegundos se encuentra por encima de la instrucción LD(DE),A más el INCremento doble con 13 microsegundos. Esta vetaja no necesita más aclaraciones.

La instrucción PUSH rr almacena el contenido de aquellos registros, que se representan con rr. Estos registros son BC, DE, HL y AF (y IX, IY).
El contador de bucles utilizado debe activarse ahora en &2000, pues el número de ejecuciones con PUSH rr se ha dividido. En lugar de &4000*1 byte, se almacenarán &2000*2.

Antes de entrar en el programa debemos eliminar una fatal posibilidad de error.
El stackpointer (SP) se coloca durante la ejecución del programa en la zona de pantalla. Para que después de la rutina pueda asignársele de nuevo su antiguo valor, Vd. debe "marcarlo".Esto se efectúa colocándolo en dos posiciones de memoria, que no pueden ser cambiados. Vamos a utilizar las posiciones de memoria &A030/31, que se encuentran en la zona superior, y que pueden intervenir en el BASIC. Después de nuestro bucle de programa y antes de la instrucción RET, el SP debe contener de nuevo su valor antiguo, para que se pueda realizar un correcto retorno al BASIC. Si esto se olvida, se bloquea inevitablemente en el sistema, lo que es nuestro mejor indicador de error de programa.
Esto ocurre porque la instrucción CALL nn, que utilizamos para llamar a nuestra rutina, asigna la dirección de retorno y algunos parámetros en el stack, de donde estos valores se toman de nuevo mediante la instrucción RET. Para ello el SP debe recibir de nuevo de forma lógica su antiguo valor, después de haber sido colocado en la zona de memoria de la pantalla.

El programa:

```
        LD   (&A030),SP  salvar SP
        LD   SP,&FFFF    stack en zona pantalla
        LD   HL,&3FFE    activar contador
        LD   BC,&FFFF    valor a almacenar
  sigue PUSH BC          asignar memoria
        DEC  HL          DECrementar contador
        BIT  5,H         activar flag
        JP   NZ,sigue    salto condicional
        LD   SP,(&A030)  actualizar SP
        RET              vuelta al programa principal
```

El bucle de este programa utiliza 18.5 microsegundos, igual que la última rutina. Pero en este caso se almacenan dos bytes al mismo tiempo, cosa que no ocurría en el caso anterior.
De este modo vemos la ventaja que aporta la instrucción PUSH.

Algunos se habrán dado cuenta de que la instrucción LD (&A030),SP direcciona sólo una posición de memoria, siendo el valor del SP de una longitud de dos bytes. Esta instrucción puede utilizarse también para este caso. En la posición &A030 se asigna el byte con el valor menor, y el valor mayor, esto es, &A031 se asigna al espacio siguiente.
Esta forma de almacenamiento de valores de dos bytes, primero el byte bajo, y luego el byte alto se aplica sin excepción. Primero se asigna siempre el valor menor (L) y luego el mayor (H). Es importante que esto se tenga en cuenta para no caer en errores.

En la instrucción LD SP,(&A030), donde la transferencia de datos se efectúa en sentido contrario, también se transfiere primero el byte L y luego el byte H.

Un ejemplo importante: Si se debe producir un salto hacia la dirección &6533, la instrucción correspondiente es CD 3365.

Para finalizar este capitulo deseo aclarar aún una rutina de borrado. Con la unión lógica OR se activa aquí el flag correspondiente, que analiza la siguiente instrucción de salto.

```
        LD    (&A020),SP   salvar el SP
        LD    SP,&FFFF     SP en pantalla
        LD    BC,&FFFF     cargar valor
        LD    HL,&C001     última dirección
        XOR   A            borrar ACCU
        LD    (&C001),A    borrar última dirección

SIGUE   PUSH  BC           almacenar valor
        OR    (HL)         finalizar test
        JP    Z,SIGUE      salto condicional

        LD    SP,(&A020)   cargar antiguo SP
        RET                vuelta al programa principal
```

Primero se asegura el valor del stack-pointer, pues es de enorme importancia, en el registro &A020. Esta posición de memoria se encuentra detrás de la rutina de borrado.

Ahora se asigna el SP a la zona de pantalla (LD SP,&FFFF), el registro BC se carga con el valor a almacenar (LD BC,&FFFF) y se lleva la dirección de la última posición de memoria a asignar, al registro HL (LD HL,&C001).

Después se carga el acumulador con 0, operando lógicamente con XOR (OR exclusiva), un método elegante y rápido que debería apuntarse, ya que muchos programas assembler utilizan este truco, a través del cual funcionan más rápidamente, pero son de más difícil comprensión.

Como el acumulador es cero, se puede seguir utilizando para borrar la posición de memoria &C001, esto es, puede asignarse 0. Esto es importante aquí porque la función de interrupción funciona impecablemente solamente si esta posición de memoria y el acumulador se inician con cero.

Si tiene dificultad para la comprensión de este apartado, debería simular la ejecución del bucle en papel, y entendería mejor el sentido de la instrucción OR (HL). De esta forma se ve directamente lo imprescindible que es el borrado previo.

En el bucle se crea la condición de salto, realizándose una comparación entre el contenido del acumulador y el de una posición de memoria. La posición de memoria se direcciona indirectamente sobre el registro HL.

El resultado de la unión OR es cero, siéndolo también el acumulador y la posición de memoria &C001. Cuando la instrucción PUSH introduce el valor &FF en la memoria &C001, el resultado de OR (HL) es distinto a cero, y la condición deja de cumplirse. Así finaliza el bucle.

Ahora vamos a tratar brevemente la necesidad de tiempo del bucle. Las instrucciones que se encuentren fuera del bucle, podemos dejarlas aparte.

```
sigue PUSH  BC        6.5
      XOR   A         3.5
      JP    NZ,sigue  5
```

Una ejecución de bucle emplea 15 microsegundos. Esto es 0.246 segundos para la pantalla completa y únicamente 7.5 microsegundos por byte.

Con el último programa podríamos haber obtenido una eficiencia máxima para la asignación de pantalla. La construcción que trabaja de modo más rápido se efectúa con una instrucción de memoria (PUSH BC) por ejecución de bucle. Si además aumentamos el número de instrucciones PUSH en el bucle, aminoramos el inconveniente que supone la fracción de tiempo consumida por JP NZ y ORA. Con 10 instrucciones PUSH por bucle, se emplean 3.7 microsegundos por byte, mientras que con un PUSH por bucle son 7.5 microsegundos. Con cada PUSH por bucle nos acercamos más al límite de 3.25 microsegundos, lo que corresponde ya a la mitad de todo un PUSH. Desgraciadamente con una técnica de este tipo nos acercamos también a la técnica de programación que llega rápidamente a los límites del ordenador, de modo que es necesario encontrar un método más razonable.

El bucle se debería aclarar de tal modo que al final sólo quedara mencionar, que es imprescindible activar el SP previamente a la instrucción RET. Así el indicador de pila se activa de nuevo en su valor inicial, esto es, se encuentra de nuevo en las posiciones de memoria, en las cuales se asignó el parámetro de salto al principio de la rutina. Un ejemplo conocido en BASIC es la instrucción GOSUB-RETURN. Si el BASIC se encuentra con una instrucción GOSUB, retiene la línea donde ésta se encontraba, para que al encontrar la siguiente instrucción RETURN "sepa", hacia qué lugar debe saltar. Esta analogía no es precisamente casual. Para este tipo de tareas, el ordenador Amstrad dispone de un BASIC-stack, que trabaja según el principio LIFO.

En el almacenamiento del contenido del registro doble no podemos disponer del valor &FFFF, que he utilizado hasta este momento por razones de claridad. Podemos tomar cualquier valor entre &01 y &FF, menos el cero. De este modo no se llegaría jamás a la condición de interrupción y se formaría un formidable lío.

El que se sienta suficientemente capaz de posibilitarlo, puede empezar. El conjunto de instrucciones es el mismo, sólo deben cambiarse un par de instrucciones. No se desanime; hasta que lleguen los primeros grandes éxitos, el ordenador se descontrolará varias veces, y nuevamente el programa aparentemente tan perfecto resulta ser una construcción errónea.

Si alguien desea incluir más instrucciones PUSH en las rutinas, ya sabe que un byte no se puede operar directamente con PUSH bajo la zona de memoria &C000. La consecuencia sería la ya suficientemente conocida caída del sistema.
Esto sucede porque la zona de memoria bajo &C000, o sea &BFFF, es la base del stack de la máquina.
Lo zona de la pila reservada al conectar el ordenador es &BF00 - &BFFF.

Para no pasar por debajo de &C000, operando con PUSH, se puede asignar p.e. un valor superior para HL. Si esto le parece poco claro, puede realizar una prueba en el "escritorio", esto es simular la ejecución del programa en papel, cosa que ayuda siempre en este tipo de problemas. Naturalmente no se debe utilizar todo el espacio de la pantalla, sino sólo una pequeña parte, para poder observar detalladamente su manera de actuar.
En este caso no debería equivocarse. Para el diseño del programa y para búsqueda de errores, este procedimiento es absolutamente practicable y eficiente.

De nuevo presentamos el programa de borrado rápido junto con un cargador BASIC. Para la instrucción de salto absoluta, he introducido una instrucción relativa, que retarda el bucle en microsegundos, haciéndose reubicable la rutina para que se pueda emplazar en cualquier lugar de la memoria.

```
5 MODE 2:SUMA=0
10 FOR I=startwert TO startwert+28
20   READ WERT$:WERT=VAL("&"+WERT$)
30   POKE I,WERT:SUMA=SUMA+WERT:NEXT
50 IF SUMA<>3682 THEN PRINT "Error en DATAs"
60 DATA ED,73,20,A0,31,FF,FF,01,FF,FF,21,01,C0,AF
70 DATA 32,01,C0,C5,B6,28,FC,ED,7B,20,A0,C9,00,00
```

13.4 Las instrucciones con más prestaciones del Z80

El microprocesador Z80 dispone de más de cien instrucciones. De entrada esto parece un gran obstáculo si se desea introducirse en esta técnica con rapidez. No se preocupe, con un análisis detallado observará que varias instrucciones pueden agruparse, ya que poseen la misma función, y sólo disponen de códigos especiales para registros concretos.

Antes de tratar algunas de las instrucciones más importantes, voy a indicar en este apartado la forma de representar la influencia de las instrucciones en el registro flag. Bajo la abreviatura del nombre del flag se encuentran caracteres propios que describen claramente la influencia sobre el flag en cuestión.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	m/t/z

*: Flag influye al correspondiente resultado
-: Flag no influído
1: Flag se activa (incondicional)
0: Flag activado de nuevo (incondicional)
?: Flag indeterminado después de operación

m: Número de ciclos de máquina
t: Número de ciclos de reloj
z: Duración de la tarea en microsegundos

b: Número del bit
r: Para el registro A, B, C, D, E, H, L

13.4.1 Instrucciones de un sólo bit

Nosotros deseamos aclarar aproximadamente 1/3 de todas las instrucciones del Z80.

Entre este tipo de instrucciones se encuentran p.e. las 168 instrucciones que tratan solamente los únicos bits de los registros A, B, C, D, E, H, L. Son las instrucciones:

BIT b,r , SET b,r y RES b,r.

La instrucción BIT b,r ya la conocemos, aunque con un enunciado distinto.

BIT b,r

Esta instrucción analiza el bit de un registro dado a partir del parámetro b.

Los parámetros b y r se encuentran en la forma general de representación de instrucciones para número del bit (b) y nombre de registro (r), donde el bit con el valor más alto de un registro, es siempre el número 7, y se encuentra en el extremo izquierdo, mientras que el bit con el valor menor siempre lleva el número 0, y se encuentra en el extremo derecho del byte. La abreviatura r significa nombre del registro de 8 bits, de los que ya no deberá preocuparse más.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	2/8/4

SET b,r

Esta instrucción activa el bit correspondiente del registro correspondiente en el valor 1. El significado del parámetro ya nos es conocido.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	2/8/4

RES b,r

La instrucción activa de nuevo el bit descrito, esto es, en el valor 0.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	2/8/4

Con ello hemos descrito ya una tercera parte de todas las instrucciones. Del mismo modo se puede explicar otra tercera parte. Las demás instrucciones necesitan más tiempo y más espacio.

De todas formas no es necesario conocer todas las instrucciones desde un principio. Estas se aprenden automáticamente cuando nos encontramos ante un problema, que no podemos resolver con los métodos que nos son hasta ese momento conocidos. En ese caso no tenemos otro remedio que informarnos sobre las posibilidades, de las que disponemos, pero que todavía no hemos aplicado.
Una descripción de este tipo sobrepasaría el marco de este libro. Por ello, recomiendo a los aficionados al assembler la literatura que se encuentra en el mercado sobre este tema específico. Si se trata sólo de aclarar las instrucciones que presentamos en este capítulo, no es necesario, ya que aquí quedan explícitamente claras.

Antes de seguir adelante, debemos nombrar todas las instrucciones que constan de un solo bit. Aún nos quedan:

```
BIT b,(HL)  BIT b,(IX+d)  BIT b,(IY+d)
```

Descripción de la instrucción:

BIT b,(HL)

El registro HL contiene aquí la dirección de la memoria (no se trata de un registro del CPU, sino sólo de un espacio de memoria), cuyo número de bit b debe ser analizado. Este direccionamiento indirecto puede ahorrar mucho tiempo al cargar.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	3/12/6

SET b,(HL) / RES b,(HL)

Estas instrucciones activan el número de bit b del registro direccionado mediante HL en 1 o en 0.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	4/15/7.5

13.4.2 Instrucciones de rotación y desplazamiento

Todas las instrucciones de rotación y desplazamiento (sólo existe una excepción que ya es muy conocida) tienen en común los siguientes modos de funcionamiento :

a) El contenido total del registro en cuestión se traslada una posición hacia la izquierda o derecha. Después de una instrucción que lleva el contenido hacia la izquierda, el contenido antiguo de bit 0 se encuentra en el bit 1, el del bit 1 en el bit 2, y así sucesivamente.

b) El último bit a activar se transporta al bit carry, lo que más adelante se tomará no como una solución necesaria por falta de espacio, sino como una excepción. En el movimiento hacia la izquierda se trata del bit 7, hacia la derecha, del bit 0.

c) Lógicamente en esta instrucción se influye siempre de forma correspondiente el flag carry.

d) Todas las instrucciones de rotación y desplazamiento son aplicables para los siguientes operandos, que se representan mediante "s":

A B C D E H L (HL) (IX+D) (IY+D)

RL s

El contenido de la posición indicada por los operandos, se traslada hacia la izquierda. Así el bit 7 llega al carry, mientras su contenido se traslada hacia el bit 0.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RR s

El contenido de la posición indicada por los operandos se traslada hacia la derecha. Así el bit 0 va al carry, mientras su contenido se traslada al bit 7.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RLC s

El contenido de la posición indicada por los operandos se traslada hacia la izquierda. Así el bit 7 pasa al bit 0, y tambien al carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RRC s

El contenido de la posición indicada por los operandos se traslada hacia la derecha. Así el bit 0 va al bit 7, y al carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SLA s

Esta instrucción desplaza hacia la izquierda el contenido de la posición indicada por S. Al bit 0 se le asigna el valor 0. El bit 7 va al carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRA s

Esta instrucción traslada hacia la derecha el contenido de la posición indicada por s. El bit 0 se traslada al carry. El bit 7 no cambia.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRL s

Esta instrucción traslada hacia la derecha el contenido de la posición s indicada por los operandos. El bit 0 se traslada al carry. El bit 7 se activa necesariamente de nuevo, esto es, se le asigna 0.

S	Z	H	P/V	N	C
*	*	0	*	0	*

Para las instrucciones de rotación y desplazamiento descritas podemos utilizar la siguiente tabla según los distintos operandos "s":

s:	m	t	z
r	2	8	4
(HL)	4	15	7,5
(IX+d)	6	23	11,5
(IY+d)	6	23	11,5

Aplicación de las instrucciones de rotación y desplazamiento:

Después de conocer algunas de las instrucciones de rotación y desplazamiento, nos dedicaremos a los ejemplos prácticos, con los que se podrá hacer una idea de su aplicación.

Algunas veces, y por causas distintas, el transmisor de datos dispone sólo de unos pocos conductores, de los cuales uno es el conductor de los datos en cuestión. A través de un conductor sólo puede pasar al mismo tiempo la información de datos de un solo bit. Esta forma de transmisión de datos, bit a bit, es la transmisión en serie.
Aquí no nos vamos a dedicar a la descripción detallada de la transmisión de datos, sino que únicamente mostraremos la forma de dividir un byte en bits separados, para que se pueda disponer de ellos en la transmisión de datos en serie. Esto se aclara con un pequeño ejemplo.

Damos por supuesto que el byte que debemos transmitir se encuentra en el registro D.

```
        LD   B,8        Contador de 8 bits por byte
sigue   XOR  A          Borrar acumulador
        SLA  D          Bit 7,D al carry
        JR   NC,M1      Saltar instrucción siguiente
        LD   A,1        Acumulador distinto a 0
M1      CALL salida     Saltar a rutina emisora
        DEC  B          Actualizar contador
        JP   NZ,sigue   Bit siguiente
        RET
```

Primero se asigna el valor 8 al contador B, para que podamos controlar el número de bits a transmitir. Luego se borra el acumulador mediante XOR A. Con SLA D se desplaza el contenido del registro D una posición hacia la izquierda, y el bit 7 se desliza al carry. Si el bit 7 contenía un 0, el bit carry es también un 0. Para que la instrucción JR NC,M1 aparezca positiva, se realiza un salto después de M1. De este modo el acumulador no se transforma, sigue siendo cero. Si hubiéramos asignado el valor 1 al bit 7, y así el carry fuera 1, siendo también la instrucción JR NC,M1 negativa, no se podría realizar el salto. Con ello, en la siguiente instrucción LD A,1, se asignaría 1 al acumulador.

En la etiqueta M1 y con ayuda de la salida CALL, se bifurcará a una rutina de salida, que debería estar constituída de tal forma, que enviara un 0 lógico, cuando el acumulador fuera distinto a cero, o un 1 lógico si el acumulador fuera igual a cero. La forma propia, con la que esto se lleva a cabo y la lógica de 1 y 0 no vamos a detallarla en este apartado.

Lo más importante es reconocer de qué manera se influye el bit carry mediante la instrucción SLA D, de modo que en base a su contenido se puede influir una instrucción de salto, y con ello crear los parámetros correspondientes para la rutina emisora.

Otro ejemplo muy claro para demostrar la aplicación del SLA r, es el de duplicar el valor de un registro o un contenido de memoria.

Primero, un ejemplo de cómo no se debería programar este tipo de multiplicación simple: 13*2

```
       LD   B,13        Multiplicando
       LD   C,2         Multiplicador
       XOR  A           Borrar acumulador
 sigue ADD  A,B         Multiplicación aditiva
       DEC  C           Actualizar contador
       JP   NZ,sigue    Salto en c<>0
       RET
```

Primero se llevan el multiplicando y el multiplicador al registro B o C. Luego se borra el acumulador, después se suma el multiplicando al acumulador. En el bucle "sigue", el acumulador se suma C veces con el valor del registro B.

El siguiente listado lleva al mismo resultado:

```
       LD   A,13       acumulador=13
       SLA  A          rotación de A hacia la izquierda
       RET
```

Vamos a demostrar la función de este tipo de multiplicación mediante su representación binaria.

```
Registro A     00001101=  8+4+1=13     (antes)
Registro A     00011010=  16+8+2=26    (después)
```

A la derecha de los números binarios se encuentran los sumandos de las posiciones binarias activadas a modo decimal. Si se corre la posición binaria hacia la izquierda, cada posición toma el valor superior siguiente, normalmente siempre el doble. Si se duplican los sumandos, se duplica también el resultado de la suma.

Pregunta:¿Qué ocurre si el número que debemos duplicar es superior a 127?

Aparece un carry. Para poder gestionar este resultado de forma ordenada, un segundo registro tiene que recoger los carrys que se presenten.
Con el siguiente ejemplo 160*4, donde se asigna 160 al registro L y se duplica dos veces, ilustraremos la actuación conjunta de dos registros.

```
        LD    H,0        Borrar reg. H
        LD    L,160      Reg. L=160
        LD    B,2        Contador
  sigue SLA   L          Desplaza L hacia izquierda
        SLA   H          Desplaza H hacia izquierda
        OR    A          Borrar carry
        DEC   B          Actualizar contador
        JP    NZ,sigue
        RET
```

Primero se borra H, se carga L con 160, y B se activa como contador.

En el bucle siguiente se desplaza siempre el registro L, pues su contenido activa el carry durante su traslado, el cual debe haber sido llevado primero a H. Como se pueden tratar los registros H y L como registros de 16 bits, obtenemos el resultado correcto, después de la ejecución de la rutina HL. La instrucción OR A se encuentra en el bucle sólo por que esté completo. Esta debe borrar un posible carry de H, pues éste se trasladaría al bit 0 del registro L, en una nueva ejecución de bucle.

Para demostrar este tipo de cálculo poco acostumbrado, el siguiente ejemplo se ha representado también en modo binario.

Registro	H	L	
	00000000	10100000	=128+32=160 (antes)
	00000001	01000000	
	00000010	10000000	512+128=640 (después)

Quien haya repasado una lista de instrucciones assembler, habrá visto que el Z80 no dispone de ninguna instrucción de rotación o de desplazamiento de 16 bits.

No es cierto, tiene una.

En el ejemplo anterior se ha substituído una instrucción de multiplicación no previa, por un desplazamiento de un registro hacia la izquierda. Giremos ahora el ejemplo, y "creemos" una instrucción de desplazamiento con una instrucción de adición.

Como ya sabemos, se puede tratar el registro doble HL casi como acumulador de 16 bits. A HL se le pueden sumar los registros BC, DE, HP y HL. Como HL se puede sumar consigo mismo, lo que corresponde a una multiplicación con el factor 2, el resultado debe corresponder a una instrucción de desplazamiento hacia la izquierda.

La instrucción de 16 bits ADD HL,HL corresponde principalmente a la instruccion de 8 bits SLA r. Los contenidos de los registros en cuestión se desplazan hacia la izquierda, el bit con mayor valor va al carry y se lleva un 0 al de valor menor.

13.4.3 Instrucciones aritméticas de 16 bits

ADD HL,rr

El contenido del registro doble señalado mediante rr se suma con el contenido de HL y se coloca en el registro HL. Los registros BC, DE, HL, y SP entran en consideración para rr. El bit half-carry se activa con un acarreo del bit 11 al bit 12, esto es, del nibble de menor valor del registro H al de mayor valor. Un acarreo de L a H no afecta al carry.

S	Z	H	P/V	N	C	
-	-	*	/	0	*	3/11/5.5

Con esta instrucción se puede demostrar de un modo simple una rotación de 16 bits no implícita en el Z80.

También se puede ejecutar una adición con carry como en el verdadero acumulador con el registro doble HL junto a una adición simple. La instrucción para ello es:

ADC HL,rr

El contenido del registro HL y del registro doble representado mediante rr se suman. A ello se le suma también el contenido del flag carry. Para rr entran en consideración los registros BC, DE, HL y SP. Se influirán todos los flags correspondientemente al resultado excepto el flag N que se activa a 0.

S	Z	H	P/V	N	C	
*	*	*	*	0	*	4/15/7.5

Con la siguiente instrucción de 16 bits, habremos descrito ya todas las posibilidades del acumulador HL.

SBC HL,rr

El contenido del registro doble señalado con rr y de la transmisión se restan del contenido del registro HL. El resultado de esta substracción se coloca de nuevo en HL. Para "rr" se pueden activar los registros BC, DE, HL, y SP. Se influirán todos los flags de forma correspondiente al resultado excepto el flag N que se activa a 1.

S	Z	H	P/V	N	C	
*	*	*	*	1	*	4/15/7.5

13.4.4 Instrucciones de bloque

Si se desea transferir posiciones de memoria consecutivas de la RAM a otra zona, dispone de las eficientes instrucciones del Z80. Son las siguientes:

LDI	Cargar con INCremento
LDIR	Cargar con INCremento y repetición
LDD	Cargar con DECremento
LDDR	Cargar con DECremento y repetición

Cada una de estas instrucciones consta en principio de más instrucciones assembler, que se ejecutan una detrás de otra.

Instrucciones de transmisión de bloque

LDI

LDI LD (DE),(HL); INC HL; INC DE; DEC BC

El contenido de la posición de memoria indicado por HL, se carga en la posición de memoria direccionada a partir de DE. Luego se INCrementan los registros dobles HL y DE. Al final se DECrementa BC.

S	Z	H	P/V	N	C	
-	-	0	*	0	-	4/16/8

LDIR

LDI LD(DE).(HL); INC HL; INC DE; DEC BC hasta BC=0

El contenido de la posición de memoria indicada por HL, se carga en la posición de memoria direccionada mediante DE. Luego se INCrementan los registros dobles HL y DE. Al final se DECrementa BC.
Esta secuencia de instrucciones se repite tantas veces como sean necesarias hasta que el registro BC se DECremente en cero.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0)
						5/21/10.5 (BC<>0)

LDD

LDI LD(DE),(HL): DEC HL; DEC DE; DEC BC

El contenido de la posición de memoria indicado por HL, se carga en el espacio de memoria direccionado por DE. Luego se DECrementan los registros dobles HL y DE. Al final se DECrementa BC.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8

LDDR

LDI LD(DE),(HL); DEC HL; DEC DE; DEC BC hasta BC=0

El contenido de la posición de memoria indicado por HL, se carga en el espacio de memoria direccionado por DE. Luego se DECrementan los registros dobles HL y DE. Al final se DECrementa BC.
Esta secuencia de instrucciones se repite hasta que el registro BC se DECrementa en cero.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0)
						5/21/10.5 (BC<>0)

Un esquema breve:

```
LDI     LD(DE),(HL); INC HL; INC DE; DEC BC
LDIR    LD(DE),(HL); INC HL; INC DE; DEC BC hasta BC=0
LDD     LD(DE),(HL); DEC HL; DEC DE; DEC BC
LDDR    LD(DE),(HL); DEC HL; DECDE; DEC BC hasta BC=0
```

Para todos los usuarios de CPC 464 ó 664, que envidien las instrucciones de bloque suplementarias del 6128, hemos presentado la siguiente rutina, que posibilita la copia de bloques de memoria.

```
LD    BC,&4000    Activar contador
LD    DE,&C000    Dirección final
LD    HL,&4000    Dirección fuente
LDIR              Instrucción de bloque
RET
```

Este programa de alto rendimiento puede introducirse con el siguiente cargador BASIC:

```
5 MODE 2:SUMA=0
10 FOR I=inicio TO inicio+11
20     READ VALOR$:VALOR=VAL("&"+VALOR$)
30     POKE I,VALOR:SUMA=SUMA+VALOR:NEXT
40 IF SUMA<>985 THEN PRINT "Error en DATAs"
50 DATA 01,00,40,11,00,40,21,00,C0,ED,B0,C9
```

Después de haber cargado la rutina con este programa, puede ejecutarla con un CALL a la dirección de inicio.
Lo único que puede aparecer es READY. La rutina ha copiado el contenido del bloque de memoria 3, esto es, el espacio de memoria de la pantalla en el bloque 1 de la RAM.
Cambie los valores subrayados de la lista DATA entre sí. Se trata de high-bytes de la dirección de inicio del bloque. Después de iniciar de nuevo el programa cargador, ejecútelo mediante CALL,y se copiará el valor de inicio del bloque 1 en la pantalla. Antes de empezar con CALL, debería borrar la pantalla con MODE 2, para poder distinguir mejor el efecto.

Puede olvidar tranquilamente el espacio de memoria donde se ha borrado, pero debería prever que no se efectúe ninguna colisión hacia abajo con los programas BASIC, y hacia arriba ningún corte con el espacio reservado para el sistema operativo. La longitud es de libre elección. La elección de la longitud de la pantalla debe obedecer sólo a cuestiones prácticas.

14. Algunas rutinas en lenguaje máquina para el tratamiento en pantalla

14.1 Scroll suave de pantalla

Cuando el ordenador lleva la salida en pantalla a la parte del borde inferior, se proporciona a sí mismo un nuevo espacio para la salida, desplazando el contenido de la pantalla en una altura de caracteres hacia arriba. La altura de los caracteres lleva una línea de pixels consigo. En ese caso el ordenador no traslada el contenido de la memoria, sino que transforma sólo el offset.

La posición de memoria cuyo contenido representa la esquina superior izquierda, ya no es &C000. Se utiliza otra posición como dirección base de la pantalla.

Con el siguiente minicargador se puede probar esta forma de scroll:

```
10 FOR I%=&A000 TO &A005
20    READ VALOR$:VALOR=VAL("&"+VALOR$)
30    POKE I%,VALOR
30    NEXT
40 DATA 06,01,CD,4D,BC,C9
```

Si se desea efectuar un scroll en la pantalla hacia abajo, deberá cambiar el valor subrayado por 0. Debería poner siempre atención en que se haya efectuado un scroll, pues pueden aparecer resultados inutilizables a partir del offset realizado.

Hemos compuesto un cargador BASIC, que realiza un scroll, y que traslada una por una cada línea de pixels. En este caso, el contenido se translada a la parte superior de la pantalla.

```
10 MODE 2:SUMME=0
20 FOR I%=&A000 TO &A036
30     READ WERT$:WERT=VAL("&"+WERT$)
40     POKE I%,WERT:SUMME=SUMME+WERT
50     NEXT
60     IF SUMME<>5695 THEN PRINT"Error en DATAs"
100 DATA 01,C7,C7,21,00,C0,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 26,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,80,FF,21,36,A0,01,50,00,ED,B0,C1
130 DATA 0E,C7,05,20,CE,C9,00,00,00
```

Si se desea efectuar scroll en el contenido de la pantalla hacia abajo, introduzca las siguientes líneas 100 y 110 de este modo:

```
100 DATA 01,C8,C7,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 29,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,00,C0,21,36,A0,01,50,00,ED,B0,C1
```

La suma de pruebas es en este caso 5699.

Si no se debe hacer scroll en todo el espacio de pantalla, sino sólo en una parte del borde izquierdo,introduzca las siguientes instrucciones después de la ejecución del programa anterior:

```
POKE &A016,10 : POKE &A02B,10
```

10 significa tamaño del espacio donde se debe efectuar scroll hacia el espacio del borde izquierdo de la pantalla.

Ahora brevemente, el programa anterior de una forma más clara y reducida.
Es más rápido pero no admite interrupciones. Se para automáticamente después del scroll. Se puede utilizar el programa cargador anterior.

El bucle FOR-NEXT se ejecuta de &A000 - &&A02F y la SUMA es 4368. Las líneas DATA se cambian por las siguientes:

```
100 DATA 01,C8,C8,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,01
110 DATA 00,08,ED,42,CB,74,20,04,01,B0,3F,09,22,00,B0,01
120 DATA 50,00,ED,B0,C1,0D,20,E1,0E,C8,05,20,D6,C9,00,00
```

14.2 Scroll de las líneas inferiores de pantalla

Para una buena representación de los componentes de un texto existen distintos medios. Por una parte la representación inversa o en color distinto de los caracteres, los avisos en forma intermitente, etc.

Un efecto nuevo e interesante es el de un aviso móvil que traslada textos o partes de textos de forma visible.

Con el siguiente programa se puede hacer scroll en la línea inferior de la pantalla hacia la izquierda, de modo que el texto que desaparece por la izquierda, aparece por la derecha.

```
A000   117702     ANF   LD    DE,&277     un proceso
A003   D5               PUSH  DE
A004   114F08           LD    DE,&84F     dirección de incio-
                                          sumando
A007   ED5341A0         LD    (&A041),DE  salvar sumando
A00B   01084F     M2    LD    BC,&4F08    contador-8-Z./&4F-Sp.
A00E   21CEC7           LD    HL,&C7CE    dirección inicio
A011   B7               OR    A           borra Carry
A012   CB16       M1          RL,(HL)
A014   2B               DEC   HL          nueva dirección
A015   05               DEC   B           final de linea?
A016   20FA             JR    NZ,M1
A018   114F00           LD    DE,&004F
A01B   3005             JR    NC,M3       tomar Carry final de
                                          linea
A01D   19               ADD   HL,DE       primera direc.anteri.
A01E   CBC6             SET   0,(HL)      activar Bit derecho
A020   1803             JR    M4          saltarse RES
A022   19         M3    ADD   HL,DE       dirección de inicio
                                          anterior
A023   CB86             RES   0,(HL)      borrar Bit derecho
```

```
A025   ED52       M4    SBC  HL,DE
A027   064F             LD   B,&4F          nuevo contad. lineas
A029   ED5B41A0         LD   DE,(&A041)     tomar sumando
A02D   19               ADD  HL,DE
A02E   0D               DEC  C
A02F   20E1             JR   NZ,M1
A031   D1               POP  DE             contador número-Sp.
A032   1B               DEC  DE
A033   CD09BB           CALL &BB09          tecla pulsada ?
A036   3808             JR   C,ENDE         interrupción
A038   D5               PUSH DE
A039   CB7A             BIT  7,D            preg.final contador
A03B   28CE             JR   Z,M2
A03D   D1               POP  DE             corrección en pila
A03E   18C0             JR   ANF            nueva vuelta
A040   C9               ENDE RET
A041                                        memoria para sumandos
A042                                        memoria para sumandos
```

No se puede hacer scroll en la pantalla, antes de solicitar esta rutina, pues de otro modo cambiaría el offset. El sistema se encuentra en MODE 2 para el sistema operativo. Los MODEs restantes que quedan sin utilizar ofrecen también efectos interesantes.

Este programa es principalmente reubicable con las instrucciones de salto relativas. Se pueden situar en cualquier parte del espacio de memoria libre de la RAM, pero le aconsejo andar con cuidado con las posiciones de memoria &A041 y &A042. En estas se encuentra almacenado el registro DE. Si se desea cargar el programa en otro lugar, deben cambiarse también de forma correspondiente estas posiciones de memoria.

El cargador BASIC:

```
10 MODE 2:SUMME=0
20 FOR I=&A000 TO &A042
30 READ WERT$:WERT=VAL("&"+WERT$)
40 POKE I,WERT:SUMME=SUMME+WERT
50 NEXT I
60 IF SUMME<>6591 THEN PRINT"Error en lineas DATA"
70 END
100 DATA 11,77,02,D5,11,4F,08,ED,53,41,A0,01,08,4F,21
110 DATA CE,C7,B7,CB,16,2B,05,20,FA,11,4F,00,30,05,19
120 DATA CB,C6,18,03,19,CB,86,ED,52,06,4F,ED,5B,41,A0
130 DATA 19,0D,20,E1,D1,1B,CD,09,BB,38,08,D5,CB,7A,28
140 DATA CE,D1,18,C0,C9,00,00,00
```

Después de iniciar el programa de carga, la rutina se encuentra ya en la memoria y puede iniciarse con CALL &A000, e interrumpirse pulsando una tecla cualquiera. El programa cargador ya no se utiliza más y puede borrarse con DELETE 1. Si protege el espacio de memoria de la rutina en lenguaje máquina mediante MEMORY &A000-1, se puede borrar el programa cargador con NEW.

Ejemplo para la aplicación de la rutina:

```
10 LOCATE 5,25 : PRINT"Repita la entrada"
20 CALL &A000
30 LOCATE 1,25 : PRINT STRING$(80," ");
40 MODE 2
50 INPUT"Nueva entrada";A
60 REM ...etc...
```

14.3 Screencopy (copia de la pantalla) para el CPC 464/664

Este apartado está pensado especialmente para aquellos usuarios del CPC 464 ó 664, que no disponen de las posibilidades de tratamiento de pantalla que se encuentran en el 6128.

Con el SCREENCOPY se puede copiar el contenido de un bloque de la RAM en otro bloque. Un bloque es una zona de memoria con una longitud de 16 Kbytes. Como la zona de la RAM del 464/664 tiene un tamaño total de 64 Kbytes, contiene 4 bloques.

	Dirección de inicio	Dirección final
Bloque 0	&0000	&3FFF
Bloque 1	&4000	&7FFF
Bloque 2	&8000	&BFFF
Bloque 3	&C000	&FFFF

Aunque tras la conexión del ordenador, el usuario dispone de más de 42000 bytes libres, puede utilizar por completo sólo el bloque 1.

Indicaciones para los bloques de memoria:

En las posiciones de memoria inferiores en el bloque 0 (aprox. 60), se encuentran las rutinas "imprescinbibles" del sistema operativo. Por lo demás, este bloque se encuentra también a su disposición.
¿Por qué "imprescindible"?
¡¡¡Introduzca el valor 0 en la posición 8 con POKE 8,0, pero sólo si no tiene ningún programa o datos importantes dentro del ordenador!!!

Como ya hemos dicho, el bloque 1 se encuentra plenamente a su disposición.

En el bloque 2 puede utilizar las posiciones de memoria &8000 - &A67B, tras la conexión del ordenador. La zona de &A67C a &BFFF (dec. 42620 - 49151) es también utilizable. Los cambios en este caso, deben efectuarse como en el bloque inferior, también con cuidado, esto significa que no debe haber ningún dato importante en el ordenador, y el diskette debe estar alejado. Este podría resultar "malparado" en caso de problemas en el sistema.

El bloque 3 contiene las informaciones de pantalla. Si se cambia su contenido, cambian también la imagen.

El modo de trabajo de la instrucción SCREENCOPY, puede observarse en el programa ejemplo, que se encuentra al final de la descripción de la instrucción en lenguaje máquina LDIR. En este programa ejemplo se copiaba la zona de memoria de pantalla en el bloque 9. Si en el bloque 1 , se encuentra almacenado previamente un contenido de imagen, puede conmutarse la imagen de pantalla en pocas décimas de segundo mediante un truco.

Aclaración:

Después de conectar, o de hacer un RESET en el ordenador, el Hardware toma las informaciones para el monitor, del bloque 3, esto es, de la zona de memoria que se inicia con la dirección &C000. Si se cambia la dirección de inicio, p.e. con el valor &4000 (primera dirección del bloque 1), la imagen de los datos de 16383 se situará en los &4000 siguientes espacios de memoria.
Allí se encuentra otro contenido de imagen, que va a ser presentado.
Con el siguiente programa se puede cambiar la base de pantalla:

```
10 FOR I=&A000 TO &A005
20    READ WERT$
30    WERT=VAL("&"+WERT$):POKE I,WERT
40    NEXT
50    DATA 3E,00,CD,08,BC,C9
60 INPUT"Que bloque  1/3",B
70 IF B=1 THEN POKE &A001,&40 ELSE POKE &A001,&C0
80 CALL &A000
```

El listado desensamblado de la rutina en lenguaje máquina es como sigue:

```
A000    3E xx       LD     A,xx
A002    CD 08 BC    CALL   BC08
A005    C9          RET
```

En la primera línea se carga el acumulador con el byte significativo de la dirección de pantalla. En el acumulador se ofrece el parámetro para la rutina SCR SET BASE, que se solicita con CALL &BC08. El RETurn no necesita comentarios.
Las instrucciones POKE de la línea 70 del programa BASIC sirven para que el high-byte de la dirección de inicio de pantalla seleccionada se POKEe en la memoria de forma que, la instrucción de carga del acumulador reciba un valor significativo.

Una variación interesante para el contenido de pantalla se puede conseguir, cuando se interrumpe el programa BASIC iniciado pulsando dos veces ESC, MODE 2 y se introduce luego la instrucción CALL &BC08.
En la pantalla aparecen más líneas de puntos. El superior representa las rutinas de sistema "imprescindibles" y las cuatro líneas restantes, represetan el programa BASIC, con el que se ha cambiado la dirección de inicio de pantalla.
Vd. ha colocado el espacio de memoria en el primer bloque (bloque 0), y observa las partes de programa que allí se encuentran.
Introduzca un par de líneas BASIC con sus números. Observe los cambios que se efectúan, al realizar la entrada con la tecla ENTER.

14.4 SCREENSWAP (intercambio de pantalla)para el 464/664

La diferencia entre SCREENCOPY y SCREENSWAP se encuentra en que con SWAP, se intercambia el contenido de las posiciones en cuestión. Con COPY, un espacio se copia en otro, siendo los dos contenidos iguales.

Como ya sabemos, para intercambiar el contenido de dos variables, necesitamos una tercera como intermediaria.
Ejemplo: Intercambie el contenido de A y B (con x como intermediario)

x=A : A=B : B=x

Si deseamos intercambiar dos bloques de memoria, donde se encuentran contenidos de imagen, disponemos de un tercer bloque completo como intermediario. Por ello, intercambiamos los contenidos en pequeñas partes. La longitud podría ser distinta, pero no muy corta, pues la rutina sería demasiado lenta. Si las partes son demasiado largas, necesitan demasiado espacio de memoria, aunque se puede reducir mediante un segundo contenido de pantalla en 1/4.

El cargador BASIC para SCREENSWAP 464/664:

```
5 START=&A000:SUMME=0
10 FOR I%=START TO START+&40
20     READ WERT$:WERT=VAL("&"+WERT$)
30     POKE I%,WERT:SUMME=SUMME+WERT:NEXT
40 IF SUMME<>5098 THEN PRINT "Error en DATAs"
50 DATA 21,00,C0,22,00,81,21,00,40,22,02,81,01
60 DATA 40,00,C5,01,00,01,11,00,80,2A,00,81,ED
70 DATA B0,01,00,01,ED,5B,00,81,2A,02,81,ED,B0
80 DATA 01,00,01,ED,5B,02,81,21,00,80,ED,B0,21
90 DATA 01,81,34,23,23,34,C1,0D,C5,20,D1,C1,C9
```

Listado assembler para SCREENSWAP 464/664:

```
A000    21 00 C0        LD    HL,&C000
A003    22 00 81        LD    (&8100),HL
A006    21 00 40        LD    HL,&4000
A009    22 02 81        LD    (&8102),HL
A00C    01 40 00        LD    BC,&0040
```

```
A00F    C5                   PUSH BC
A010    01 00 01    sigue    LD   BC,&0100
A013    11 00 80             LD   DE,&8000
A016    2A 00 81             LD   HL,(&8100)
A019    ED B0                LDIR
A01B    01 00 01             LD   BC,&0100
A01E    ED 5B 00 81          LD   DE,(&8100)
A022    2A 02 81             LD   HL,(&8102)
A025    ED B0                LDIR
A027    01 00 01             LD   BC,&0100
A02A    ED 5B 02 81          LD   DE,(&8102)
A02E    21 00 80             LD   HL,&8000
A031    ED B0                LDIR
A033    21 01 81             LD   HL,&8101
A036    34                   INC  (HL)
A037    23                   INC  HL
A038    23                   INC  HL
A039    34                   INC  (HL)
A03A    C1                   POP  BC
A03B    0D                   DEC  C
A03C    C5                   PUSH BC
A03D    20 D1                JR   NZ,sigue
A03F    C1                   POP  BC
A040    C9                   RET
```

El programa se encuentra en forma reubicable. De todas formas, no puede colocarse en cualquier lugar. Por un lado, no puede estar en el bloque 1 de la memoria, pues éste es utilizado como memoria adicional de la pantalla. Por otro, el intermediario para el SWAP y los espacios de memoria reservados, están directamente detrás del bloque 1 en el espacio de &8000 - &8103. Con ello se dan los espacios de memoria &4000 - &8103 como espacio para el programa.

Si en la línea 5 del cargador BASIC coloca la dirección de inicio en &8104 conseguirá un espacio abierto (&4000 - &8144), donde se encuentran todas las posiciones de memoria necesarias para el SWAP.

15. Un interface simple de BASIC para los registros en lenguaje máquina del Z80

Todos sabemos que en la programación, existen algunos casos en que es mejor utilizar rutinas en lenguaje máquina del CPC que las instrucciones BASIC correspondientes.
Un buen ejemplo es la rutina, que se solicita mediante CALL &BB06. Esta rutina espera la siguiente pulsación de una tecla, que se puede activar con cualquiera de las teclas. Luego el programa vuelve de nuevo al programa BASIC.
La rutina &BB06 no necesita parámetros de salto.

Con la instrucción SPEED WRITE x podemos elegir entre los parámetros 0 y 1, la velocidad de escritura para la cassette, estando la velocidad máxima conectada en 1. Así la salida está activada en 2000 baudios. Este es naturalmente el máximo que se puede conseguir con el CPC.
Las transmisiones superiores se consiguen mediante la rutina, seleccionada sobre &BC68. Para ello, se deben activar concretamente 3 registros del Z80. Con el siguiente programa se pueden cargar los registros de una forma simple.

```
10 MODE 2:SUMME=0:GOSUB 100
20 FOR I=1 TO 10
30   READ REG$,PLATZ$:PRINT REG$;:INPUT" ";WERT$
40   IF WERT$="" THEN WERT$="00"
50   WERT=VAL("&"+WERT$):PLATZ=VAL("&"+PLATZ$)
60   POKE &A000+PLATZ,WERT
70   NEXT
80 CALL &A000:END
100 FOR I=&A000 TO &A012:READ W$:W=VAL("&"+W$)
110 POKE I,W:SUMME=SUMME+W:NEXT
120 IF SUMME <>960 THEN PRINT "Error en DATAs":END
130 RETURN
140 DATA 21,0,0,E5,F1,1,0,0,11,0,0,21,0,0,CD,0,0,C9,0
150 DATA A,2,F,1,B,7,C,6,D,A,E,9,H,D,L,C,HB,10,LB,F
```

El bucle en las líneas 100 y 110 junto con los datos de la 140 ofrecen el interface en forma de una pequeña rutina assembler a partir de la posición de memoria &A000.

El programa solicita los valores que deben cargarse en los registros A, F, B, C, D, E, H, y L. Las siguientes entradas HB y LB se refieren al high-byte y low-byte de la rutina elegida. Las entradas que no interesan, pueden saltarse con ENTER.

Con este instrumento tan simple, tenemos una buena posibilidad para el BASIC, utilizando el Firmware del CPC. Volvamos a la velocidad de la cassette. Los parámetros de salida son la longitud para la mitad de un bit nulo en HL y la longitud del análisis previo en el registro A.

1000 baudios:	HL=333	A=25
2000 baudios:	HL=167	A=50

Estos valores ofrecen una alta seguridad en la transmisión de datos. El aumento de la velocidad de carga no depende sólo del material de la cinta.

16. Almacenamiento de programas en lenguaje máquina en la memoria

El método más utilizado para almacenar programas en lenguaje máquina, seguramente ya le es conocido. Los valores leídos a partir de líneas data, se escriben sobre la RAM del BASIC en la memoria. Primero debe reservarse el espacio mediante MEMORY. Este método tiene una desventaja:

Como las definiciones de variables se alamacenan también sobre el final del BASIC, se efectúan superposiciones. Los programadores del sistema conocen este problema, y han diseñado una protección. Si el límite superior de memoria se coloca mediante MEMORY, debajo del final de la definición de símbolo, todas las instrucciones "SYMBOL-AFTER" que le sigan, llevarán el aviso de error "inproper argument". Con ello se evita que con el aumento del espacio de símbolos, se sobreescriban los programas en lenguaje máquina registrados eventualmente. Por una parte, esto es ideal para la protección de programas en lenguaje máquina, pero por otra, cada "SYMBOL AFTER" provoca un aviso de error. Esto significa p.e. que se interrumpe continuamente la ejecución del programa, cuando aparece "SYMBOL AFTER". Una posibilidad sería p.e. introducir desde un principio, "SYMBOL AFTER 0" en modo directo.

Para programas permanentes, esto no es una solución muy positiva.

Una segunda posibilidad es colocar la instrucción en la primera línea y borrarla después de la primera ejecución.

Es aplicable en el CPC 464:

```
10 SYMBOL AFTER 100
20 a=PEEK(&AE81)+256*PEEK(&AE82)
30 POKE a+4,&C5
40 FOR i=a+5 TO a+PEEK(a)+256*PEEK(a+1)-2:POKE i,32:NEXT
50 MEMORY &A000
                .
                .
                .
```

Para el CPC 664/6128:

```
10 SYMBOL AFTER 100
20 a=PEEK(&AE64)+256*PEEK(&AE65)+1
30 POKE a+4,&C5
40 FOR i=a+5 TO a+PEEK(a)+256*PEEK(a+1)-2:POKE i,32:NEXT
50 MEMORY &A000
                .
                .
                .
```

Otro método, que permite introducir definiciones de símbolo en programas, es cambiar el puntero himem interno, para la aceptación de "SYMBOL AFTER". Si utiliza este método, debe procurar en todo caso que no se escriba encima de sus programas en lenguaje máquina con definiciones de símbolo.

Es aplicable en el CPC 664 y PC 6128:

```
10 MEMORY &9FFF
20 REM ....
30 oldhimem=PEEK(&AE5E)+256*PEEK(&AE5F)
40 POKE &AE5E,PEEK(&AE60):POKE &AE5F,PEEK(&AE61)
50 POKE &B735,0
60 SYMBOL AFTER 200
70 MEMORY oldhimem
80 REM ....
```

Para el 464:

```
10 MEMORY &9FFF
20 REM ....
30 oldhimem=PEEK(&AE7B)+256*PEEK(&AE7C)
40 POKE &AE7B,PEEK(&AE7D):POKE &AE7C,PEEK(&AE7E)
50 POKE &B295,0
60 SYMBOL AFTER 200
70 MEMORY oldhimem
80 REM ....
```

Como todos estos métodos son sólo instrucciones, tiene aún la posibilidad de utilizar otros, en según que casos de forma ventajosa.
Primero le presentaremos dos posibilidades, que en principio utilizan el mismo procedimiento, pero que se almacenan en una zona de memoria distinta.

En lugar de almacenar programas en lenguaje máquina por encima de la zona destinada al BASIC, podemos hacerlo por debajo de la memoria de programas BASIC.

El "inicio de un programa BASIC menos un puntero" se encuentra en la dirección &AE81/2 (en 664/6128: &AE64/5). Normalmente indica la dirección &16F, esto es, los programas se inician en la dirección &170.

Si a este puntero le asignamos un valor grande, al que llamaremos "Lowmem", dispondremos del espacio &170 hasta Lowmem para programas en lenguaje máquina. O sea:

```
POKE &AE64,INT(Lowmem/256)
POKE &AE65,LOWMEM-INT(LOWMEM/256)*256
NEW
```

(464: para &AE69/5 tenemos 6AE81/2)

NEW se debe introducir directamente al final, ya que de otro modo, los punteros, para la tabla de variables p.e., no se activan de modo correcto.

El traslado de los espacios no es visible en una programación BASIC. Del mismo modo funciona también "SYMBOL AFTER". El espacio se unifica debajo del programa BASIC, siendo muy apropiado para el almacenamiento de programas en lenguaje máquina.

Los programas en lenguaje máquina se pueden almacenar en el sitema de la RAM pero con ciertas excepciones. Un espacio utilizable de la RAM de este tipo es la memoria reservable key. Su dirección de inicio se encuentra en la dirección &B4E1/2 (664/6128: &B62B/C). Normalmente la tabla empieza en la dirección &B446 (664/6128: &B590); ocupa 152 bytes. La desventaja se encuentra en que la reserva KEY de teclas no es reutilizable. Para que esto no ocurra, las teclas de ampliación deberían reactivarse con KEY DEF en su función normal.

También es posible un almacenamiento de la tabla de caracteres definidos por el usuario, y se pueden pasar por alto las definiciones de caracteres que no se van a utilizar de nuevo. El código de la primera tabla de símbolos de los caracteres tratados, está en la dirección &B294 (664/6128: &B734). La magnitud resulta de:

```
PRINT (256-PEEK(&B194))*8  para 464
PRINT (256-PEEK(&B734))*8  para 664/6128
```

Si ya se han definido previamente deben mantener los códigos de caracteres 32 - 127.

La dirección de inicio de la tabla se encuentra en la dirección &B296/7 (664/6128: &B736/7), y la dirección final en &B096/7 (664/6128: &B075/6). Finalmente también es posible el almacenamiento de un vídeo-RAM (dirección &C000 hasta &FFFF). Para ello, la siguiente indicación:

El vídeo-RAM tiene una magnitud de 16K (16K=16*1024=16384). En el MODE2 se encuentran almacenados "sólo" 25*80*8 puntos. La diferencia es que no se utilizan 384=8*48 bytes.

Al conectar, o después de la instrucción MODE, tenemos ya 48 bytes sin utilizar en un espacio de &C7D0-&C7FF, &CFD0-&CFFF, &D7D0-&D7FF, etc. El problema de este método es que no se puede hacer nunca(!) scroll en la pantalla.

Los espacios libres resultantes se trasladarían, y se escribiría encima de los programas. De todos modos esto puede ser para el almacenamiento de algunos programas, un buen espacio, ya que no presenta más excepciones.

Hemos descrito un par de métodos más, que funcionan sólo en algunos programas en lenguaje máquina. Todos los programas en lenguaje máquina, que deben ser almacenados, deben ser trasladables (relocativos), esto significa que no pueden contener direcciones absolutas, que se encuentren en el espacio de direccionamiento del mismo programa. En algunos casos y con un poco de suerte, se pueden escribir programas en lenguaje máquina, que no contengan direcciones absolutas, y que p.e. trabajen con saltos relativos dentro del programa. Los programas deben tener estas características, para que el programa pueda tratarlos. Con ello se garantiza sólo que no se cambie ningún byte del programa, aunque la dirección de inicio sigue siendo variable y puede cambiar según las circunstancias

¿Cómo es posible?

Muy simple: El BASIC permite permanentemente la salida de la administración de cadenas de bytes, o sea, de strings. Un string consta de una serie de códigos, uno detrás de otro. Su longitud puede llegar hasta 255 caracteres. Consecuentemente podremos almacenar un programa en lenguaje máquina (=serie de códigos) a modo de string (cadena).

Con un bucle se leerán los códigos de las líneas DATA y se sumarán con CHR$ de la serie detrás de un string.

```
MAPRO$=MAPRO$+CHR$(byte)
```

Así se ha almacenado el programa en lenguaje máquina.

La llamada se realiza mediante la función @, que indica el string-descriptor (ver ayuda de programación). Para nuestro programa que está en MAPRO$, podemos escribir:

```
CALL PEEK(@MAPRO$+1)+256*PEEK@MAPRO$+2)
```

La segunda posibilidad, constituída de este modo, permite el almacenamiento correspondiente a la magnitud del programa.

Esta vez utilizaremos simplemente una variable de campo de tipo entero. Un número entero consta de byte high y low. En un campo se almacenarán simplemente los valores que corresponden al indice, uno detrás de otro. En este sentido, podemos utilizar campos numéricos INT para el almacenamiento de programas en lenguaje máquina. Más información sobre variables de campo y su tratamiento interno, puede encontrarla en el capítulo 22. El siguiente programa aclara este principio. Es importante:

1) El campo debe ser de tipo entero (INT) (%)
2) Si el programa tiene una longitud inexacta, es imprescindible introducir &00 en las líneas DATA.

```
10 lang=26: 'Numero de Bytes en el programa
20 DIM meldung%(INT((lang-1)/2))
30 FOR i=0 TO INT ((lang-1)/2)
40 READ l$,h$
50 meldung%(i)=VAL("&"+h$+l$)
60 NEXT i
70 END
80 DATA 3e,01,cd,0e,bc,cd,15,b9
90 DATA 7c,21,57,86,fe,01,28,06
100 DATA 2e,5c,38,02,2e,77,cd,0b
110 DATA 00,c9
120 REM Llamada con CALL @meldung%(0)
```

Tras la entrada de RUN, se puede solicitar el programa en todo momento con CALL @AVISO%(0). El indice dado en la solicitud debe ser 0.

¿Ha probado el programa? Quizás en algún momento haya pensado que el ordenador se ha estropeado o se ha llevado a cabo un Reset.

No es así. Unicamente ha aparecido el aviso de conexión. El pequeño programa "aviso" es interesante en la medida que reconoce automáticamente una versión correcta del ordenador. Esto puede observarse mediante la rutina de sistema "KL Version Number".

Listado assembler del programa:

```
A000             10                 ;Mensaje
A000 3E01        20          LD     a,1
A002 CD0EBC      30          CALL &bc0e ; SCR Modo a
A005 CD15B9      40          CALL &b915 ; KL número de versión
A008 7C          50          LD     a,h
; Versión BASIC 0,1 ó 2
A009 215786      60          LD     hl,&8657 ; para 664
A00C FE01        70          CP     1
A00E 28FE        80          JR     z,ok
A010 2E5C        90          LD     l,&5c ; para 464
A012 38FE       100          JR     c,ok
A014 2E77       110          LD     l,&77 ; para 6128
**** Linea  80  : OK=&A016
**** Linea  100  : OK=&A016
A016 CD0B00     120 OK       CALL &b
A019 C9         130          RET

Programa : mensaje
Inicio : &A000    Fin : A019
Longitud : 001A
 0 Error
Tabla de variables :
OK   A016
```

17. Almacenamiento de rutinas assembler y espacios de memoria

Con el excelente interpretador BASIC del CPC puede almacenar directamente zonas de la RAM. La sintaxis de esta forma de la instrucción SAVE es:

```
SAVE "nombre del programa",B,direc. de inicio,longitud
```

De este modo se pueden almacenar rutinas en lenguaje máquina en un diskette o en una cinta. Ejemplo:

El programa a almacenar se encuentra en &A000 a &A013.

```
SAVE"DEMONOMBRE",B,&A000,&14
```

Observe que la entrada no sea demasiado corta. En el ejemplo, la longitud de la rutina es &14 byte, aunque el último byte tenga el número &xx13. Si falta el último byte, que suele ser RET,se bloquea al programa después de llamado.

A parte de la posibilidad de almacenar rutinas assembler, pueden SAVE (salvarse) contenidos de imagen totales.

```
SAVE "SCREEN",B,&C000,&4000
```

En este ejemplo se almacena una zona de memoria continua a partir de la dirección &C000 con la longitud &4000 como fichero binario bajo el nombre de SCREEN.
Ahora se ofrecen buenas posibilidades, para utilizar las ya mencionadas rutinas para el screencopy, screenswap o la conmutación de la dirección base de pantalla.

Como el procedimiento de carga de una imagen completa, dura algunos segundos, se ofrece la posibilidad de cargar la imagen primero en el bloque 1 de la RAM, y luego, llevarlo rápidamente al espacio de pantalla, con una de las instrucciones.

```
LOAD "SCREEN",&4000 : CALL screencopy
```

18. Rutinas útiles del sistema operativo

Dirección &0000: RST 0 - RESET

La solicitud de esta rutina mediante CALL 0 funciona como una conexión/desconexión del ordenador.

Dirección &0008: RST &08 - Low Jump

Esta rutina salta a una dirección del sistema operativo ROM o a la RAM superior. Los bits 14 y 15 determinan la selección ROM/RAM. Un bit activado significa RAM y un bit no activado ROM. Bit 14 determina la dirección de la zona inferior (&0-&3FFF) y bit 15 la superior (&C000 a &FFFF).

Dirección &000C: JP (HL) con selección ROM/RAM

Salto direccionado indirectamente a la dirección indicada por el registro HL. Los bits 14 y 15 de HL funcionan como RST &08.

Dirección &0010: RST &10 : Side Call

Sirve para solicitar una rutina en una expansión de ROM.

Dirección &0018: RST &18 - Far Call

Sirve para solicitar una rutina en alguna parte de la ROM o la RAM. Detrás de la instrucción RST &18 se encuentra la dirección de un vector, que debe contener la dirección final y el status de la ROM/RAM.

Dirección &0020:   RST &20 - RAM Lam LD A,(HL)

El acumulador se carga con el valor de la dirección, en HL. A través de esta rutina se selecciona siempre RAM.

Dirección &0028:   RST &28 - Firm Jump

Sirve para la llamada de una rutina en el sistema operativo (Firmware). De este modo la dirección aparece directamente detrás de la instrucción.

Dirección &0030:   User Restart

Esta rutina está a disposición de los programas propios.
En el capítulo 23 encontrará un ejemplo de aplicación del RST &30.

Dirección &0038:   Salto interrupt

A partir de esta dirección se solicitan las rutinas de interrupción.

Dirección &BB06:   KM(Key Manager) - Wait Char

El código ASCII de las teclas pulsadas pasa al acumulador.

Dirección &BB15:   KM EXP Buffer

Esta rutina determina el espacio de la RAM, reservado para la definición de teclas de función. Como este espacio ha sido determinado con poca magnitud, puede aumentarse con esta rutina.

Para ello, se introduce la dirección de inicio en el registro DE y la longitud del buffer en el registro HL. Si se define una longitud suficiente, puede asignarse una cadena de hasta 32 caracteres para cada tecla de función. Las entradas anteriores pueden borrarse con la ejecución de esta rutina.

Dirección &BB24: KM Get Joystick

El registro H contiene el estado del Joystick 1, el registro L, el del 2.

Dirección &BB39: KM Set Repeat.

La función Repeat se conecta para la tecla con el número contenido en el acumulador, en el caso de que B<>0, si B=0, se desconecta.

Dirección &BB3C: KM Get Repeat

El flag Z se activa, si la tecla con el número contenido en el acumulador no se encuentra en Repeat, en caso contrario, mantiene su valor.

Dirección &BB3F: KM Set Delay

Entrega del número de tecla: A, Tiempo de retardo en el registro H, velocidad de repetición en el registro L.

Dirección &BB42: KM Set Delay

Entrega: Número de tecla en A
Devolución: H:Tiempo de retardo, L: Velocidad de repetición.

Dirección &BB5A: TXT Output

Asigna el carácter en pantalla cuyo valor código corresponde al valor del acumulador.

Dirección &BB60:   TXT Read Char

Lectura de un carácter de pantalla. En HL se entrega la posición del carácter (H-línea / L-columna). Si se reconoce un carácter válido, se activa el carry y se carga el código ASCII del carácter en el acumulador.

Dirección &BB6C: TXT Clear Window

Borra la ventana de pantalla actual.

Dirección &BB75:   TXT Set Cursor

H/L equivale a línea/columna.

Dirección &BB78:   TXT Get Cursor

Dirección &BB81:   TXT Cursor on

Dirección &BB84:   TXT Cursor off

Dirección &BB90:   TXT SET PEN

El color Pen se asigna al número INK contenido en el acumulador.

Dirección &BB93: TXT GET PEN

Lectura del número PEN-INK en el acumulador

Dirección &BB96: TXT SET PAPER (s. PEN)

Dirección &BB99: TXT GET PAPER (s. PEN)

Dirección &BB9C: TXT INVERSE

Dirección &BBA5: TXT GET MATRIX

La dirección de la definición de código A se entrega de nuevo en el registro HL. El carry está activado, si se trata de un carácter definido por el usuario.

Dirección &BBA8: TXT SET MATRIX

La matriz del carácter definido por el usuario con el código A, se carga con la matriz a partir de la dirección HL.

Dirección &BBC0: GRA Move Absolute

El registro DE contiene la coordenada X, el registro HL, la coordenada Y.

Dirección &BBC3:   GRA MOVE relative

El registro DE contiene la coordenada relativa X, el registro HL la coordenada relativa Y.

Dirección &BBC6:   GRA ask Cursor

Registro reservado para el absolute move.

Dirección &BBC9:   GRA SET PEN A=ink

Dirección &BBE1:   GRA GET PEN A=ink

Dirección &BBE4:   GRA SET PAPER A=ink

Dirección &BBE7:   GRA GET PAPER A=ink

Dirección &BBEA: GRA Plot Absolut

Dirección &BBED: GRA Plot Relative

DE: relativo,coordenada X
HL: relativo,coordenada Y

Dirección &BBF0:   GRA TEXT Absolut

DE: coordenada X
HL: coordenada Y
Devolución: A contiene INK del punto

Dirección &BBF3:   GRA Text relativ

Dirección &BBF6:   GRA DRAW LINE

La línea es trazada desde cursor gráfico actual a las coordenadas finales.

Dirección &BBF9:   GRA DRAW LINE relative

Dirección &BBFC: GRA WRITE Char

El carácter A aparece en la posición del cursor.

Dirección &BC0B: Set location

Se introduce la dirección de inicio actual del carácter superior izquierdo de pantalla. A contiene el high byte de la dirección base (normalmente &C0) y HL el offset de la dirección base para la dirección de inicio.

Dirección &BC0E: SCR Mode A

Se conecta el modo de pantalla correspondiente al contenido del acumulador.

Dirección &BC11:   SCR GET MODE

El número MODE actual se carga en el acumulador. Así se influyen los flags:

MODE 0: C
MODE 1: Z
MODE 2: NC

Dirección &BC14: SCR Clear screen con ink 0

Dirección &BC1A: SCR CHAR POS

Entrada: H-línea / L-columna de la posición de líneas
Devolución: HL: Dirección del primer byte
B: Amplitud del carácter en el byte (1, 2, o 3)

Dirección &BC1D: SCR DOT POS

Entrada DE: coordenada X / HL coordenada Y
Devolución : HL: Dirección de pantalla
C: máscara de bits, que toma sólo en consideración el punto en cuestión.
B: Número de puntos -1 por byte (1,2, o 3)

Dirección &BC20: SCR NEXT BYTE

Incrementar la dirección de pantalla HL en un byte hacia la derecha.

Dirección &BC23: SCR PREV BYTE

Trasladar la dirección de pantalla HL en un byte hacia la izquierda.

Dirección &BC26

La dirección de pantalla HL se posiciona en la siguiente línea.

Dirección &BC29: SCR PREV LINE

Coloca la dirección de inicio HL en la línea anterior.

Dirección &BC32:   SET INK Color

A: número INK; color B y C.

Dirección &BC35:   SCR GET INKcolor (como arriba)

Dirección &BC38:   SCR SET Bordercolor (B,C)

Dirección &BC3B: SCR GET Bordercolor (B,C)

Dirección &BC3E: SCR SET Flash time

H: Tiempo para primer color
L: Tiempo para segundo color

Dirección &BC41:   SCR GET Flash time

Dirección &BC5F: SCR línea horizontal

A: INK, DE:inicio X, BC: final Y, HL: coordenada Y

Dirección &BC62:   CR línea vertical

A: INK, DE: coordenada X, BC: final Y, HL: inicio Y

Dirección &BC9B: CAS (o Disk) Catalog

Dirección &BCD1:   KL (Kernel) Log Ext

Conecta las ampliaciones RSX

Dirección &BD0D: KL Time please

Ofrece de nuevo el valor del timer como valor de 4 bytes en DE y HL

Dirección &BD10:   KL Time Set

Dirección &BD2B: MC Print Char

Envía el valor del acumulador a la impresora

Dirección &BD2E: MC Printer Busy

Analiza si la impresora está dispuesta para la impresión. Si no es así se activa el carry flag

Dirección &BD37:   Jump Restore : Parada de emergencia

Reactiva todos los vectores de salto en su valor de salida.

Dirección &BDD3:   Write

El contenido del acumulador aparece en pantalla como carácter ASCII.

19. Rutinas útiles del interpretador BASIC

El orden de direcciones significa:

Dirección 1 para 464, dirección 2 para 664 y dirección 3 para 6128

Dirección &C356: &C3A0: &C3A3: Print

Envía el carácter correspondiente al contenido del acumulador por el canal actual. El número de canal se encuentra en la posición de memoria &AC21, &AC06, &AC06. La salida de nuestro programa XREF/DUMP se puede imprimir p.e. con POKE &AC06,8:|XREF (464: POKE &AC21,8:|XREF).

Dirección &C34E: &C398: &C39B: Line Feed

Manda line-feed por el canal actual.

Dirección &C43C: &C472: &C475: BREAK test

Examina si se ha imprimido ESC. Si es así, la rutina, salta al modo de espera (como ya conocemos en BASIC).

Dirección &CA94: &CB55: &CB58: Salida de error

En la entrega del número de error en el acumulador (en 664 y 6128 en el registro E), aparece el correspondiente aviso de error y se interrumpe el programa.

Dirección &D5DB: &D619: &D61C:
Buscar puntero de la tabla de variables-caracteres

La letra de inicio de la variable a buscar se entrega en el acumulador. Como devolución, BC contiene la dirección de inicio de la tabla de variables y HL la dirección, donde se encuentra almacenado el puntero a la primera variable con la letra indicada.

Dirección &D6B3: &D6EC: &D6EF:
Buscar puntero de la tabla de variables

Para la entrega, HL debe indicar el inicio de una variable en el programa, dicho de modo más exacto, al carácter de reconocimiento de tipo, del que cada variable dispone. Si la variable se encuentra en la tabla, aparecerá su dirección de inicio. En caso contrario, se introducirá la variable en la lista y se dará de nuevo la dirección de inicio.

Dirección &DD37: &DE25: &DE2A: CHR NEXT

Examina si el byte siguiente del programa BASIC es igual al byte, que sigue a la solicitud de la rutina. En caso de desigualdad, aparece un aviso de error.

Dirección &DD3F: &DE2C: &DE31: CHRGET

La rutina sirve para tomar el byte siguiente. Primero se incrementa HL. Se lee el byte que se encuentra en la dirección HL. Si se trata de un espacio vacío (code 32), se lee el byte siguiente y así sucesivamente.
EL primer carácter "no vacío" se dará en el acumulador.

Si se trata de un byte nulo, se activa el flag Z. La rutina se emplea en la lectura de programas BASIC. Allí, el byte nulo significa fin de una línea.

Dirección &DD51: &DE3D: &DE42: CHRGOT

Lee de nuevo el último byte repasado, y examina si se ha llegado al final de la instrucción. Si es así, C está activado.

Dirección &DD55: &DE41: &DE46: CHKKOMMA

Analiza si el byte actual representa una coma. Si es así, se lee el byte siguiente y se activa el carry. Sinó, el carry-flag, mantiene su valor.

Dirección &E8FF: &E9B9: &E9BE: BASIC DO Rutina BC

Esta rutina ejecuta el programa BASIC actual y salta después de la lectura del inicio de cada línea a una rutina cualquiera de la dirección BC. Allí se puede buscar la línea. Después de tratar la línea se salta de nuevo con RET y automáticamente se pasa a la línea siguiente.

Dirección &E943: &E9FD: &EA02: SKIP COMMAND

En relación con la última rutina tratada, esta rutina salta una unidad de instrucción en una línea.

Dirección &EE79: &EF44: &EF49: PRINT HL decimal

Para la salida del número de línea de un programa BASIC, se puede utilizar esta línea. Sólo se debe entregar el número de línea al registro HL como valor de 2 bytes.

Dirección &F236: &F2D5: &F2DA: PRINT FAC

Esta rutina ofrece el valor contenido en ese momento en el FAC en forma decimal.

Dirección &FF4B: &FF6C: &FF6C: VAR FAC

Copia el valor de una variable en el FAC. Para ello, HL debe contener la dirección del valor.

Dirección &FF71: &FF92: &FF92 TEST letra

El contenido del acumulador se interpreta como ASCII y si es posible, se transforma en letras mayúsculas. Si no se trata del código ASCII de una letra, el carry se encuentra reactivado.

Dirección &CFEE: &D058: &D055: Operand Missing

Dirección &C205: &CB30: &C210: Improper Argument

20. Compatibilidad entre los tres CPCs

En este capítulo deseamos indicar dónde se encuentran las diferencias o "semejanzas" más importantes entre los tres ordenadores Amstrad CPC 464, CPC 664, y CPC 6128 dentro del tema de la programación en lenguaje máquina.

Básicamente podemos decir que la ROM de los tres ordenadores es casi la misma. Esto significa desgraciadamente que existen casi todas las rutinas en los tres ordenadores. El problema es que casi todas las rutinas tienen las direcciones directas de salto distintas. Unicamente y por suerte, la mayoría de vectores de salto están en la RAM central. Los vectores de salto especiales son los siguientes:

- HIGH KERNEL
  Direcciones de &B900 a &B920

- Para KEY-Manager
  Direcciones &BB00 a &BB4D

- Para TEXT-Pack
  Direcciones &BB4E a &BBB9

- Para GRAPHIC-Pack
  Direcciones &BBBA a &BBFE

- Para SCReen-Pack
  Direcciones &BBFF a &BC64

- Para CAS- o DISK-Manager
  Direcciones &BC65 a &BCA6

- Para SOUND-Pack
  Direcciones &BCA7 a &BCC5

- Para KERNEL (Low)
  Direcciones &BCC8 a &BD12

- Para MaChine-Pack
  direcciones &BD13 hasta &BD35

- igual que el vector JUMP RESTORE &BD37

Desgraciadamente no podemos analizar todas las rutinas mencionadas, ya que no hemos encontrado ninguna rutina en este campo que tenga la misma función en los tres ordenadores Amstrad.

Después del vector JUMP RESTORE en la dirección &BD37, termina la compatibilidad.

En el 464 aparece la llamada al Line-Editor y se salta luego a las rutinas aritméticas. En el 664 y 6128, aparecen algunos nuevos vectores de salto para p.e. la llamada a la rutina Fill.
El vector LINE EDITOR se encuentra en la dirección &BD3A, y luego aparecen las rutinas aritméticas.
Los vectores Jump (vectores, que sólo se pueden solicitar dentro de sistemas operativos conectados) de la dirección &BDCD hasta la dirección &BDF3 son iguales en todas las versiones de ordenadores.

Si desea saltar directamente alguna rutina, como p.e. las rutinas BASIC, las reglas de dirección de salto son distintas en las tres versiones, aunque entre el CPC 664 y el CPC 6128 las diferencias son mínimas.

Otra diferencia importante es que las indirecciones del interpretador BASIC, que se encuentran en el CPC 664 y el CPC 6128 han sido substraídas totalmente por cuestión de espacio. Con ello se han desplazado también todas las direcciones de datos de sistemas ("PEEKs y POKEs") por encima del 664. Estas direcciones son prácticamente iguales en el CPC 664 y 6128.

Resultado:

Si es posible, deberían utilizarse los vectores mencionados para la programación. Si no es posible, no tendrá más remedio que escribir cada vez tres versiones distintas. La única rutina clara dentro de este lío de direcciones, es la rutina KL VERSION, que por suerte, sinó todo sería en vano, se encuentra en una dirección normalizada (&B915). Esta rutina ofrece de nuevo el número de la high ROM actual. Así H contiene el número de la ROM (1 para BASIC-ROM) y L el número de versión 0=CPC 464, 1=CPC 664 y 2=CPC 6128.

Con esta rutina se ha realizado el programa "aviso" del capítulo que trataba la entrada de programas en lenguaje máquina, ejecutable en los tres ordenadores.

21. Ampliación de instrucciones con RSX

En el BASIC-Amstrad es posible llevar a cabo una ampliación de instrucciones, de maneras distintas. En el 464 se pueden ampliar las rutinas ROM disponibles a través de las "zonas Patch" en la RAM. Esta posibilidad está relativamente limitada pues sólo exiten 9 de estos posibles patches. En el CPC 6128 no se han tenido en cuenta por cuestiones de espacio. En todos los CPC existe un método, que posibilita la ampliación con instrucciones propias. Todos conocen este método. Se utiliza para todas las instrucciones de diskete AMSDOS.

Seguramente habrá realizado ya una solicitud del CP/M-Modus con el |CPM, especialmente si dispone de una unidad de discos. La instrucción CPM empieza con una barra ("|"). En la entrada de "CPM" (sin barra) obtendrá sólo un aviso "syntax error". Si Vd. dispone de un ordenador sin floppy (es posible sólo con el 464), obtendrá después de |CPM un "Unknown Command".

Se trata de una instrucción, de la que solamente se dispondrá en el ordenador, si éste está conectado a un floppy. Así se constituye como ampliación.

La barra "|" comunica al sistema que se trata de una instrucción de ampliación. El método, para ampliaciones eventuales únicamente con signos especiales que se convierten en señales, para que el ordenador los reconozca y los ejecute, se encuentra ya incorporado en el ordenador. A esta forma de enmarcar las instrucciones se le llama RSX. RSX significa "Resident System Extension", que es tanto como "Ampliación residente del sistema". El RSX está pensado para incluir nuevas instrucciones, que se encuentran en la ROM suplementaria (Expansions ROM). Para el sistema operativo del floppy existe este tipo de expansions ROM, donde se encuentran también las instrucciones |CPM y |ERA.

También podemos utilizar el método RSX para incluir nuestras propias rutinas como instrucciones reales, y así utilizarlas de modo confortable.

21.1 DOKE - escribir valor de dos bytes en la memoria

Vamos a aclarar el funcionamiento del método RSX con el ejemplo de la instrucción DOKE.

DOKE es casi una instrucción de "doble POKE". Con POKE se puede escribir una posición de memoria con un valor entre 0 y 255. DOKE escribe dos posiciones una detrás de otra con un valor entre 0 y 65535. Para ello el valor se divide en high-byte y low-byte. El low-byte se coloca en la posición inferior de la memoria y el high en la posición superior. Con DOKE se simplifica el cambio de varios parámetros del sistema, que se encuentran almacenados en la RAM (usualmente como PEEKs y POKEs).

La entrega de parámetros en el RSX es análoga a la de la instrucción CALL.

La instrucción DOKE necesita dos parámetros: La dirección a partir de la cual el valor debe almacenarse, y el valor, que se debe colocar a partir de esa dirección. Así la instrucción tiene el formato siguiente:

```
|DOKE, dirección, valor
```

Los valores se entregarán de la siguiente forma:

A: Contiene el número de parámetros dados

Flags: Zero-flag=1 si no se ha entregado ningún parámetro, de lo contrario 0

B: Contiene 32-número de parámetros

DE: Contiene los últimos parámetros entregados

IX: Dirección del último elemento. El penúltimo elemento se encuentra en la dirección IX+2, el siguiente en IX+4, etc.

La rutina DOKE es:

```
100 ' LD L,(IX+2) ; cargar en el registro HL
110 ' LD H,(IX+3) ; dirección entregada
120 ' LD (HL),E ; almacenar low-byte
130 ' INC HL ; activar dirección en high-byte
140 ' LD (HL),D ; almacenar high-byte
150 ' RET ; listo!
```

Con CALL &A000,dirección,valor podemos solicitar la instrucción DOKE, siempre que se encuentre almacenado a partir de &A000. Pruébelo:

```
CALL &A000,..,..
```

Con ello disponemos de una rutina DOKE funcional. Esta llamada con CALL es un poco molesta. DOKE debe solicitarse ahora también con DOKE. Luego se debe escribir una rutina para incluir el comando DOKE. El trabajo principal lo realiza la rutina "LOGEXT" del sistema operativo, que realiza la inclusión propiamente dicha. Unicamente debemos entregar algunos valores en el Logext.

Para incorporar DOKE, el sistema debe "saber" lo siguiente:

1) Nombre de la ampliación
2) Dirección de la rutina
3) Dirección de 4 bytes no utilizados, que se emplean internamente en el tratamiento de la rutina

La dirección de los cuatro bytes de sistema se carga antes de la llamada de Logext en el registro HL. Aparte de esto se carga BC con la dirección de inicio de una tabla, donde se encuentran las direcciones siguientes.

Esta tabla contiene primero la dirección de una segunda tabla, donde están uno o más nombres de distintas rutinas a incluir. A la dirección de la segunda tabla en la primera, le sigue una instrucción de salto hacia la rutina, o rutinas a unir. Si se incluyen más rutinas, las instrucciones de salto y los nombres de las rutinas deben corresponderse.

La segunda tabla contiene como ya hemos dicho, los nombres de las rutinas. Allí, el bit 7 de la última letra de un nombre debe estar siempre activado, para poder separar un nombre de otro.

Volvamos a nuestro ejemplo:

```
10 ' LD BC,RSXTAB ; dirección de la primera tabla
20 ' LD HL,SYSBYT ; dirección del byte de sistema
30 ' CALL &BCD1 ; Llamada rutina Logext
40 ' RET ; Incluir lista
50 ' RSXTAB Dw NAMTAB ; Tabla 1 contiene como primera direc-
     ción la dirección de inicio de la 2 dirección (tabla de
     nombres)
60 ' JP START ; y luego instrucciones de salto hacia la
     rutina
70 ' NAMTAB DM "DOK" ; tabla de nombres
80 ' DB &c5 : =ASC("E")+128 para conectar bit 7
90 ' DB 0 ; byte nulo señala el final de la tabla de nombres
95 ' SYSBYT DS 4 ; reservar 4 bytes para kernal
100 ' START LD....aquí empieza la rutina
                 :                       :
```

Después del ensamblado del programa completo (esto es, cargador BASIC) se incluirá la instrucción DOKE con CALL &A000. A partir de este punto, DOKE es una instrucción de ampliación (RSX), esto significa que la instrucción se puede utilizar como cualquier otra, en modo directo, o en el programa.

Observe que la rutina de inclusión se solicite sólo una vez. Los 4 bytes de sistema se utilizan para mostrar las demás tablas RSX eventuales.

Si se inicializa esta rutina por segunda vez, el indicador que debe mostrar la siguiente tabla RSX, indica naturalmente la misma tabla.

Con ello, puede ocurrir que el reconocimiento de instrucciones se encuentre en un bucle sin fín, que no nos favorece en absoluto. El siguiente pequeño programa demuestra cómo se puede efectuar una nueva llamada, escribiendo al pricipio de la rutina Init, una instrucción RET.

```
A000 010000      10  INIT    LD     bc,rsxtab
A003 210000      20          LD     hl,sysbyt
A006 CDD1BC      30          CALL   &bcd1
A009 3EC9        40          LD     a,&c9
A00B 3200A0      50          LD     (init),a
A00E C9          60          RET
**** Linea   10  : RSXTAB=&A00F
A00F 0000        70  RSXTAB DW      namtab
A011 C30000      80          JP     START
**** Linea   70  : NAMTAB=&A014
A014 444F4B       90  NAMTAB DM      "DOK"
A017 C5         100          DB     &c5
A018 00         110          DB     0
**** Linea   20  : SYSBYT=&A019
A019            120  SYSBYT DS      4
**** Linea   80  : START=&A01D
A01D DD6E02     130  START  LD      l,(ix+2)
A020 DD6603     140          LD     h,(ix+3)
A023 73         150          LD     (hl),e
A024 23         160          INC    hl
A025 72         170          LD     (hl),d
A026 C9         180          RET


Programa : doke
Inicio : &A000    Final : &A026
Extensió : 0027
 0 Error
Tabla de variables
INIT   A000  RSXTAB A00F  NAMTAB A014  SYSBYT A019
START  A01D
```

21.2 RPEEK - Lectura a partir de la RAM o la ROM

El siguiente listado muestra cómo se utiliza la función @ junto con RSX o CALL.
En nuestro caso, se introduce la dirección de inicio del valor de una variable entera con @, y luego se entrega al programa en lenguaje máquina. El valor obtenido se escribe luego por el programa en la dirección entregada y así está a su disposición en el BASIC bajo la variable en cuestión.

El programa siguiente implementa una instrucción PEEK ampliada con ayuda de RSX. Con esta nueva instrucción es posible la lectura de direcciones tanto en la RAM como en la ROM y en las ROM de expansión (p.e. diskettes ROM).

La instrucción tiene el siguiente formato:

```
|RPEEK,@variable entera, dirección, status
```

P.e. mediante

```
|RPEEK,@w%,&C000,-7
```

Se carga en la variable w% el valor que se encuentra en la primera dirección de la ROM de diskette. Es importante que w% se utilice al menos una vez antes, sinó @w% causaría el error "improper argument". Para el status sirve la siguiente tabla:

1: RAM
2: ROM
0, -1, -2 hasta -251 seleccionar la ROM de expansión

Todos los demás valores para el status provocan un "improper argument".

```
A000              10                  ; ampliación con RSX
A000              20
; "¦RPEEK,@Intvar.,Adr.,Status"
A000              30                  ; Satus    1 :RAM
A000              40                  ;          2 :ROM
A000              50                  ; 0,-1,...,-251 : Exp. ROM
A000              60
A000              70  AOPMIS EQU  &d055
; 464 : &cfee / 664 : &d058
A000              80  AIMPAR EQU  &c21d
; 464 : &c205 / 664 : &cb50
A000 010000       90 INIT     LD   bc,rsxtab
A003 210000      100          LD   hl,sysbyt
A006 CDD1BC      110          CALL &bcd1
A009 3EC9        120          LD   a,&c9
A00B 320000      130          LD   (init),a
A00E C9          140          RET
**** Linea   90   : RSXTAB=&A00F
A00F 0000        150  RSXTAB DW   namtab
A011 C30000      160          JP   start
**** Linea   150   : NAMTAB=&A014
A014 52504545    170  NAMTAB DM   "RPEE"
A018 CB          180          DB   &cb
A019 00          190          DB   0
**** Linea   100   : SYSBYT=&A01A
A01A             200  SYSBIT DS   4
A01E DF          210  OPMIS  RST  &18
A01F 0000        220          DW   vek1
A021 C9          230          RET
**** Linea   220   : VEK1=&A022
A022 55D0        240  VEK1   DW   aopmis
A024 FD          250          DB   253
A025 DF          260  IMPARG RST  &18
A026 0000        270          DW   vek2
A028 C9          280          RET
**** Linea   270   : VEK2=&A029
A029 1DC2        290  VEK2   DW   aimpar
A02B FD          300          DB   253
```

```
**** Linea    160   : START=&A02C
A02C FE03        310  START  CP    3 ; dos parámetros
A02E 20EE        320         JR    nz,opmis
; no, pues Operand missing
A030 7A          330         LD    a,d
A031 FE00        340         CP    0
A033 28FE        350         JR    z,noexro
; Status>0,  luego no rom de expansión
A035 FEFF        360         CP    &ff ; suma<255 ?
A037 20EC        370         JR    nz,imparg
; si, luego argumento impropio
A039 7B          380         LD    a,e
A03A ED44        390         NEG
A03C FEFC        400         CP    252
A03E 30E5        410         JR    nc,imparg
A040 18FE        420         JR    setsta
**** Linea    350   : NOEXRO=&A042
A042 7B          430  NOEXRO LD    a,e
A043 FE03        440         CP    3
A045 30DE        450         JR    nc,imparg
A047 C6FE        460         ADD   a,254
A049 30FE        470         JR    nc,setsta
A04B D604        480         SUB   4
**** Linea    420   : SETSTA=&A04D
**** Linea    470   : SETSTA=&A04D
A04D 320000      490  SETSTA LD     (status),a
A050 DD6E02      500         LD    l,(ix+2)
A053 DD6603      510         LD    h,(ix+3)
A056 DF          520         RST   &18
A057 0000        530         DW    vektor
A059 DD6E04      540         LD    l,(ix+4)
A05C DD6605      550         LD    h,(ix+5)
A05F 77          560         LD    (hl),a
A060 97          570         SUB   a
A061 23          580         INC   hl
A062 77          590         LD    (hl),a
A063 C9          600         RET
**** Linea    530   : VEKTOR=&A064
A064 0000        610  VEKTOR DW    anfang
```

```
**** Linea   490  : STATUS=&A066
A066 FD         620  STATUS DB   253
**** Linea   610  : ANFANG=&A067
A067 7E         630  ANFANG LD   a,(hl)
A068 C9         640         RET


Programa : rpeek
Inicio : &A000    Final : &A068
Extensión : 0069
 0 Error
Tabla de variables :
AOPMIS D055  AIMPAR C21D  INIT   A000  RSXTAB  A00F
NAMTAB A014  SYSBYT A01A  OPMIS  A01E  VEK1    A022
IMPARG A025  VEK2   A029  START  A02C  NOEXRO  A042
SETSTA A04D  VEKTOR A064  STATUS A066  ANFANG  A067
```

```
10 FOR i=&A000 TO &A068
20 READ a$:w=VAL("&H"+a$)
30 s=s+w:POKE i,w:NEXT
40 IF s<> 17592 THEN PRINT"Error en DATAs":END
50 '464: IF s<> 17720 THEN ...
60 '664: IF s<> 17655 THEN ...
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,1A,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,14
100 DATA A0,C3,2C,A0,52,50,45,45
110 DATA CB,00,20,A0,0F,A0,DF,22
120 DATA A0,C9,55,D0,FD,DF,29,A0
130 '464:..,..,ee,cf,..,..,..,..
140 '664:..,..,58,d0,..,..,..,..
150 DATA C9,1D,C2,FD,FE,03,20,EE
160 '464:..,05,c2,..,..,..,..,..
170 '664:..,50,cb,..,..,..,..,..
180 DATA 7A,FE,00,28,0D,FE,FF,20
190 DATA EC,7B,ED,44,FE,FC,30,E5
200 DATA 18,0B,7B,FE,03,30,DE,C6
210 DATA FE,30,02,D6,04,32,66,A0
220 DATA DD,6E,02,DD,66,03,DF,64
230 DATA A0,DD,6E,04,DD,66,05,77
240 DATA 97,23,77,C9,67,A0,FD,7E
250 DATA C9
```

22. La estructura de variables de campo

En este capítulo nos ocuparemos brevemente del almacenamiento interno de variables de campo. Como aplicación práctica le presentaremos un programa, con el que puede calcular la suma de todos los elementos en lenguaje máquina.

Las variables de campo se almacenan como variables normales en una tabla detrás del programa BASIC. La dirección de inicio de la tabla se encuentra en la dirección &AE87 (664/6128: &AE68). Como las variables de campo necesitan mucho espacio de memoria, este se debe reservar previamente con la instrucción DIM. Las variables de campo se reconocen como las variables normales, por la dirección de encadenamiento después de las letras de inicio incorporadas en listas encadenadas.

La estructura de la tabla de variables de campo es:

Primero tenemos dos bytes, que contienen la dirección de encadenamiento. Después sigue el nombre de la variable, donde como ya sabemos, se encuentra sumado el código ASCII de la última letra con 128 = &80, estando el código activado en bit 7. Tras el nombre sigue la cifra de reconocimiento de tipo. En este aspecto la variable de campo se parece a una variable normal.

A continuación sigue la longitud de toda la entrada siguiente, que será almacenada como número de dos bytes. Después el número de índices del campo, y para cada índice del campo tenemos su tamaño máximo.

Tras estas informaciones, siguen la serie de todos los valores de los campos. Para ello, en un campo INTeger se prevén dos bytes para cada entrada, que contienen el valor por sí mismos. En variables reales, la longitud de una entrada es de cinco bytes. En este caso el valor se encuentra también representado. En campos de cadena se almacena el descriptor de cadena de tres bytes de longitud.

Observemos de nuevo la estructura:

2 bytes	Dirección de encadenamiento
n bytes	Para la representación del nombre
1 byte	Cifra de reconocimiento de tipos
2 bytes	Longitud total de la siguiente entrada
1 byte	Número de índices
cada 2 bytes	Para valor máximo de cada índice
m bytes	Valor de cada elemento del campo
	Longitud 2 para INTeger
	Longitud 3 para String
	Longitud 5 para REAL

El siguiente programa utiliza esta estructura, para llevar a cabo la suma de todos los elementos de un campo numérico. Para el formato de la instrucción, observe el siguiente listado Assembler.

```
A000             10                      ; suma de un campo
A000             20                      ; ofrece suma todos valores
A000             30                      ; a partir de 1 varia. campo
A000             40
A000             50
;   Formato:     "call    &a000,@<nombre   de   variable(indices
0,...)>,número indices,@<variable para resultado>
A000             60
A000             70            ORG  &a000
A000 DF          80            RST  &18
A001 0000        90            DW   vektor
A003 C9         100            RET
**** Linea  90  : VEKTOR=&A004
A004 0000       110    VEKTOR DW    sumar
A006 FD         120            DB   253 ; UROM on
A007            130
A007            140
; Adr. para 6128 ; 464 , 664
A007            150    IMPARG EQU   &c21d
; &FA9C , &cb50  improper arg.
A007            160    TYPMIS EQU   &ff62
; &FF40 , &ff62  type mismatch
A007            170    VARHND EQU   &ff8c
; &FF66 , &ff87  Variable (hl) tras (de)
A007            180    VARFAC EQU   &ff6c
; &ff4b , &ff6c  Variable (hl) tras FAC
A007            190    CPHLDE EQU   &ffb8 , &ffd8  CP hl,de
A007            200    READHD EQU   &bd7c
; &bd58 , &bd79  Real arithm. (hl)+(de)
A007            210    FAC    EQU   &b0a0 ; &b0c2 , &b0a0
A007            220    TYP    EQU   &b09f ; &b0c1 , &b09f
A007            230
**** Linea  110  : SUMAR=&A007
A007 D5         240    SUMAR  PUSH de
A008 210000     250           LD    hl,FAC1 ; los 5 Bytes
A00B 0605       260           LD    b,5 ; de FAC1 en
A00D 3600       270    NEXT   LD    (hl),0 ; cero
A00F 23         280           INC   hl
```

```
A010 10FB     290           DJNZ next
A012 DD6E04   300           LD   l,(ix+4)
A015 DD6605   310           LD   h,(ix+5)
; dirección del primer elemento después de HL
A018 DD7E02   320           LD   a,(ix+2) ; número indice
A01B 47       330           LD   b,a
A01C E5       340           PUSH hl ; dirección primer elemento
A01D 87       350           ADD  a,a ; número indice * 2
A01E C601     360           ADD  a,1 ; de 2 Bytes
A020 1600     370           LD   d,0 ; para cada index
A022 5F       380           LD   e,a ; partiendo de hl
A023 B7       390           OR   a ; borrar Carry
A024 ED52     400           SBC  hl,de ; HL indica el número
A026 7E       410           LD   a,(hl) ; del indice
A027 B8       420           CP   b ; comparar
A028 C21DC2   430           JP   nz,imparg
; si el número de indice falso: "argumento impropio"
A02B 2B       440           DEC  hl ; extensión del campo
A02C 2B       450           DEC  hl ; saltar
A02D 2B       460           DEC  hl ; HL indica luego
A02E 7E       470           LD   a,(hl) ; signo identif. tipos
A02F C601     480           ADD  a,1
A031 FE03     490           CP   3 ; si aparece cadena, luego
A033 CA62FF   500           JP   z,typmis ; Type mismatch
A036 4F       510           LD   c,a
A037 329FB0   520           LD   (TYP),a
A03A 23       530           INC  hl
A03B 5E       540           LD   e,(hl) ; extensión del
a03C 23       550           INC  hl ; campo
A03D 56       560           LD   d,(hl) ; en Bytes para HL
A03E 19       570           ADD  hl,de ;sumar= Fin de un campo
A03F E3       580           EX   (sp),hl
; intercambiar inicio con final
A040 0600     590           LD   b,0
A042 F3       600           DI     ; porque es más rápido
A043 E5       610    WEITER PUSH hl ;direc. elemento siguiente
A044 C5       620           PUSH bc ; Tipo en c
A045 CD6CFF   630           CALL varfac ;var(hl) después de FAC
A048 21A0B0   640           LD   hl,fac
```

```
A04B 110000    650            LD   de,fac1
A04E FE05      660            CP   5 ; Real ?
A050 28FE      670            JR   z,real
A052 7E        680            LD   a,(hl)
A053 23        690            INC  hl ; tomar valor
A054 66        700            LD   h,(hl) ; del elemento actual
A055 6F        710            LD   l,a
A056 EB        720            EX   de,hl
A057 7E        730            LD   a,(hl) ; tomar
A058 23        740            INC  hl ; valor de la suma
A059 66        750            LD   h,(hl)
A05A 6F        760            LD   l,a
A05B 19        770            ADD  hl,de
A05C 220000    780            LD   (FAC1),hl ; FAC1=hl
A05F 18FE      790            JR   break
**** Linea  670  : REAL=&A061
A061 EB        800     REAL   EX   de,hl
A062 CD7CBD    810            CALL readhd ; real (hl)=(hl)+(de)
**** Linea  790  : BREAK=&A065
A065 C1        820     BREAK  POP  bc ; Tipo en c
A066 E1        830            POP  hl ; activar HL en dirección
A067 09        840            ADD  hl,bc ; siguiente
A068 D1        850            POP  de ; dirección final
A069 D5        860            PUSH de
A06A CDD8FF    870            CALL cphlde ; comparar hl con de
A06D 38D4      880            JR   c,weiter
A06F FB        890            EI    ; fin constitución suma
A070 D1        900            POP  de
A071 210000    910            LD   hl,FAC1
A074 D1        920            POP  de ; copiar resultado direc.
A075 CB8CFF    930            CALL varhnd ; final
A078 C9        940            RET
**** Linea  250  : FAC1=&A079
**** Linea  650  : FAC1=&A079
**** Linea  780  : FAC1=&A079
**** Linea  910  : FAC1=&A079
A079           950     FAC1   DS   5
```

programa : campo

```
Inicio : &A000     Final : &A07D
Extensión : 007E
 0  Error
Tabla de variables :
VEKTOR A004   IMPARG C21D   TYPMIS FF62   VARHND FF8C
VARFAC FF6C   CPHLDE FFD8   READHD BD7C   FAC    B0A0
TYP    B09F   SUMAR  A007   NEXT   A00D   WEITER A043
REAL   A061   BREAK  A065   FAC1   A079
```

```
10 REM BASIC cargador para 6128
20 REM campo de suma del programa
30 FOR i=&A000 TO &A07D
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 15294 THEN PRINT"Error en DATAs":END
70 PRINT"ok!":END
80 DATA DF,04,A0,C9,07,A0,FD,D5
90 DATA 21,79,A0,06,05,36,00,23
100 DATA 10,FB,DD,6E,04,DD,66,05
110 DATA DD,7E,02,47,E5,87,C6,01
120 DATA 16,00,5F,B7,ED,52,7E,B8
130 DATA C2,1D,C2,2B,2B,2B,7E,C6
140 DATA 01,FE,03,CA,62,FF,4F,32
150 DATA 9F,B0,23,5E,23,56,19,E3
160 DATA 06,00,F3,E5,C5,CD,6C,FF
170 DATA 21,A0,B0,11,79,A0,FE,05
180 DATA 28,0F,7E,23,66,6F,EB,7E
190 DATA 23,66,6F,19,22,79,A0,18
200 DATA 04,EB,CD,7C,BD,C1,E1,09
210 DATA D1,D5,CD,D8,FF,38,D4,FB
220 DATA D1,21,79,A0,D1,CD,8C,FF
230 DATA C9,06,08,0D,20,CC
```

```
10 REM Campo de suma 464
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15372 THEN PRINT"Error en DATAs":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,9C,FA,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,40,FF,4F,32
140 DATA C1,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,4B,FF
160 DATA 21,C2,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,58,BD,C1,E1,09
200 DATA D1,D5,CD,B8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,66,FF
220 DATA C9,06,08,0D,20,CC
```

```
10 REM Campo de suma para 664
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15346 THEN PRINT"Error en DATAs":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,50,CB,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,62,FF,4F,32
140 DATA 9F,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,6C,FF
160 DATA 21,A0,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,79,BD,C1,E1,09
200 DATA D1,D5,CD,D8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,87,FF
220 DATA C9,06,08,0D,20,CC
```

23. Programas en lenguaje máquina móviles: Reubicación

23.1 ¿Para qué sirve la reubicación?

Casi todos los que han utilizado programas en lenguaje máquina -especialmente ampliaciones de instrucción-, más de una vez, ya sean escritos por uno mismo o tomados de otra persona, conocen este problema:
Cada programa tomado por sí sólo se ejecuta libre de objeciones. Pero, si se desea combinar estos programas entre sí, es decir, disponer de ellos al mismo tiempo, aparecen los problemas. Uno de estos graves problemas producidos al combinar ampliaciones de instrucciones, resulta de la reubicación de vectores. Ello conlleva, que la última ampliación cargada, vuelve a anular las variaciones efectuadas en los vectores por la primera ampliación cargada, para el reconocimiento de instrucción. Afortunadamente, este problema no existe en el CPC, gracias a la posibilidad de las ampliaciones RSX (RSX = Resident System Expansion). Mediante la aplicación del mecanismo RSX se pueden combinar prácticamente cualquiera de las ampliaciones de instrucciones, porque no es necesario reubicar los vectores del sistema operativo, para reconocer las instrucciones de ampliación; el reconocimiento de instrucciones externas ya está previsto en el sistema operativo.

El otro problema que se produce en la combinación de programas escitos en lenguaje máquina no deriva del sistema operativo, sino de una particularidad del lenguaje máquina: en principio, un programa de lenguaje máquina está ligado a una zona de memoria concreta, para la que ha sido escrito. Aunque se pueda encontrar en cualquier zona de la memoria. solamente es realizable en el lugar que se le ha asignado. Esto sucede porque muchas instrucciones en lenguaje máquina indican direcciones de memoria absolutas. En las instrucciones que se refieren a direcciones dentro del lenguaje máquina, se produce una consecuencia lógica, y es que estas instrucciones sólo pueden encontrar en las direcciones nombradas la parte de programa o los datos, a los que deben acceder, si el programa se encuentra en el espacio de memoria correcto.

Como las ampliaciones de instrucciones se encuentran normalmente en el final superior de la memoria disponible, para gastar el mínimo del valioso espacio de memoria para los programas BASIC, se produce el difícil problema en la combinación de ampliaciones de instruciones: Normalmente estos colisionan en la asignación del espacio de memoria, donde una ampliación cargada, borra como mínimo una parte de una ampliación cargada anteriormente. La consecuencia fatal de esta colisión es como máximo el bloqueo del ordenador, y como mínimo una función errónea de la ampliación de instrucción cargada.

23.2 ¿Cómo funciona generalmente la reubicación?

La palabra mágica para solucionar las dificultades presentadas hasta este momento en la combinación de programas en lenguaje máquina, es "reubicación". Detrás de esta palabra se esconde una técnica que hace que un programa en lenguaje máquina sea independiente como mínimo de la dirección absoluta del programa. Para conseguirlo, existen distintas posibilidades:

La primera de estas posibilidades se refiere a la aplicación de ficheros intermedios especiales, que contienen un código cargador especial, y que junto al propio programa contienen también informaciones importantes para la reubicación. Este código cargador no es ejecutable como programa máquina. Primero se debe llevar a un espacio concreto de memoria mediante un programa cargador especial, y allí se transforma automáticamente en un programa ejecutable en ese espacio de memoria. Estos programas cargadores utilizados para esta tarea no son normalmente simples programas cargadores, sino un llamado programa "link". Un programa "link" no se encuentra sólo en situación de cargar e iniciar programas, sino que además une automáticamente dentro de un programa general, una serie de programas-módulo.
(De aquí deriva su nombre inglés "to link" = "unir")

En este sentido trabaja p.e. el MACRO80-assembler de la empresa Microsoft. El programa "link" correspondiente, además de poder elegir si el programa iniciado ha de almacenarse como prgrama máquina en diskette o simplemente en la memoria, ofrece la posibilidad de unir automáticamente rutinas de una biblioteca. Tampoco le representará ningún problema la intervención de un programa módulo en una parte o en los datos de otro programa módulo cargado simultáneamente; esto se llevará a cabo mediante las llamadas "etiquetas globales", que se administran prácticamente de forma completa por el programa "link".

Dentro de las posibilidades que se ofrecen, este método es sin duda el mejor. Aquí disponemos de posibilidades, cuya flexibilidad y versatilidad están prácticamente limitadas sólo por el espacio de memoria y la velocidad del ordenador. Con este paquete de software se pueden estructurar y construir de forma modular, grandes programas máquina, que son de tratamiento más simples y más propicios para los cambios.
De este modo, es posible p.e. colocar programas y datos en espacios de memoria concretos, separados unos de otros, lo que es muy práctico en el caso de que se desee copiar un programa en una EPROM.
Las considerables ventajas de esta solución se deben pagar de un modo u otro: Se debe disponer de un assembler especial para crear el programa cargador. Este assembler no es más completo que otro normal, pero ya que sus posibilidades son considerables, también es considerable la dificultad que representa el utilizarlo de modo provechoso. El "link" no es tampoco un programa simple.

La segunda posibilidad parte de un programa escrito con un ensamblador totalmente normal. Este programa se coloca en una dirección base cualquiera, con el ensamblador. (Para analizar el programa aconsejamos una dirección ejecutable en el mismo).

Ahora empieza el trabajo duro: El programa debe hacerse reubicable, se debe proveer con un programa cargador especial asignado al programa, que solicita la dirección base deseada, o comunica y cambia automáticamente entre la carga y la inicialización del programa, las direcciones que se relacionan dentro del programa para un cierto fin.
El programa cargador tiene en principio siempre el mismo aspecto y puede escribirse en BASIC sin problemas, aunque se debe fabricar la tabla de las direcciones aceptadas por separado, para cada programa, y se debe cambiar prácticamente casi en cada cambio efectuado. Esta tabla es ya muy extensa al principio del programa, y crece drásticamente con la longitud del programa.
La confección automática de esta tabla es posible, p.e. con un programa auxiliar, que relaciona sus informaciones con comentarios especiales en el código fuente assembler, aunque el diseño del programa auxiliar es relativamente realizable.

Existen otros métodos que tienen un carácter más experimental: Un "reubicador automático" casi desensambla un programa a tratar y protagoniza en caso necesario un cambio de la dirección, dependiendo de que una instrucción, que pudiera contener una dirección absoluta (instrucciones de carga de 16 bits, instrucciones absolutas de salto con direccionamiento directo ampliado), se relacione con una dirección dentro del programa. Este método es cómodo en su aplicación, pero extraodinariamente dudoso: Mientras en el programa tratado exista una tabla de direcciones que se refiera al mismo, no funcionará. Aparte de esto, un valor que se encuentre detrás de una instrucción de carga de 16 bits, que corresponda a una dirección dentro del programa, no será siempre una dirección; los valores de inicio de contador y otras constantes se cambiarán por estas reubicaciones sin consideración en ese caso.

Por estas razones, en programas de una extensión considerable, se puede considerar sólo como casualidad, el hecho de que se produzca un resultado útil.

23.3 También puede hacerse más fácil

Los dos métodos de reubicación presentados en el apartado anterior son verdaderamente brillantes, - lo que no se puede decir, por suerte, del tercero. Estos dos métodos tienen, de todos modos, desventajas decisivas. El primer método lleva consigo la nueva creación de programas no muy económicos, y el tratamiento unilateral en un nuevo sistema de la programación de ensamblado. El segundo método conlleva un costoso "trabajo manual".

El método que deseamos presentarle no tiene prácticamente nada en común con los tres anteriores. En este caso podrá observar un método de reubicación, que con un gasto mínimo (el programa máquina que utilizaremos en este caso tiene una extensión de sólo 42 bytes) puede llevar a cabo todas las tareas de reubicación.

El principio, sobre el que se basa este método, es tan simple como genial: El problema de la inmovilidad de programas máquina sería insignificante, si existiesen equivalentes a las instrucciones relativas de salto (JR) también para las instrucciones JP y CALL (con una distancia de salto de 16 bits - se entiende -, para poder acceder a la memoria completa) así como para las instrucciones de carga con direccionamiento ampliado. Por ello hemos pensado: Si estas instrucciones son tan necesarias, debemos construir una. El "invento" de nuevas instrucciones es algo que no se puede hacer diariamente, por ello, describiremos de forma más exacta el procedimiento.
Para la construcción de nuestras nuevas instrucciones, nos servimos de un método, que han diseñado los constructores del Z80, el método Prefix.

Se trata de dar a un Opcode previamente definido mediante la presentación de este llamado Prefix, un nuevo significado. De este modo, de una instrucción relativa al registro HL, se pasa a una instrucción para el regitro IX mediante el prefijo &DD, y mediante &FD a otra para el registro IY. A través de nuestro nuevo prefijo se debe cambiar una instrucción cualquiera, con un operando de 16 bits, de modo que no se tome el operando de forma absoluta, sino que se relacione de forma relativa con la situación del registro contador del programa (PC). En este sentido las partes o datos de un programa pueden corresponder relativamente dentro del programa con la instrucción que se desea constituir, mediante lo cual, la dirección absoluta del programa no se considera en esta intervención.

Naturalmente no se puede cambiar de modo simple la función del procesador Z80 para introducir nuestras instrucciones correspondientes. En lugar de ello, necesitamos un pequeño programa auxiliar (como hemos dicho: 42 bytes), que se solicita con la aparición de nuestro prefijo de reubicación, y reacciona de la forma correspondiente. Como prefijo utilizaremos la instrucción libre RST &30 (corresponde a la función de un CALL &0030) que se encuentra en el sistema operativo del CPC. También podríamos emplear una instrucción CALL, aunque tiene considerables desventajas en la velocidad de tratamiento y en la utilización del espacio. La ventaja de la instrucción RST es que al contrario de la instrucción CALL, en una dirección de salto funciona prácticamente sin indicaciones, y asigna sólo un byte.
La instrucción RST &30 se añadirá simplemente antes de la instrucción que debe interpretarse de modo relativo. Antes de ejecutar la propia instrucción en el programa, el procresador llega al RST &30 y ejecuta el subprograma correspondiente. El subprograma saca de la instrucción, a la que pertenece, la dirección relativa (el lugar donde se encuentra la instrucción, lo indica la dirección de salto de retorno de la instrucción RST que se encuentra en la pila), calcula a partir de ésta y de la dirección base (que es igual que la dirección de salto de retorno), la dirección efectiva, introduce esta instrucción, solapa la instrucción RST, a la que ha llamado con una instrucción NOP, y vuelve de nuevo al programa, donde se lleva a cabo la propia instrucción.

Mediante este método, el programa se encuentra unido de nuevo a su espacio de memoria mientras se ejecute, y así es prácticamente imposible que el programa pueda moverse durante su utilización en la memoria. Si fuera de otro modo, necesitaríamos un extenso programa auxiliar, que naturalmente absorbería más tiempo de cálculo. Un programa de este tipo, a diferencia del programa aplicado en este caso, debería reconocer las instrucciones en cuestión y llevar a cabo su función correspondiente. Nuestro programa provee la instrucción sólo con direcciones correctas y permite al procesador, ejecutar la instrucción por sí mismo. La considerable ventaja de velocidad de nuestro programa no se debe únicamente al hecho de que el programa auxiliar no sólo pueda renunciar al reconocimiento y simulación del comando siguiente a la instrucción RST, sino sobre todo a que éste comienza a actuar en la primera ejecución de la instrucción; después de la instrucción tratada se encuentra la dirección absoluta correcta, y la instrucción se puede ejecutar de forma completamente normal. El mismo tiempo de cálculo se necesita para la instrucción NOP, que se encuentra delante del comando y se trata de forma muy rápida.

23.4 El programa de reubicación

De momento ya hemos discutido bastante esta oscura teoría. Observe un momento el listado de nuestro programa auxiliar. En la siguiente descripción de funciones, seguramente se le aclararán las bases teóricas de este método.

Listado del programa auxiliar de reubicación:

```
0000   E3     RELRST   EX     (SP),HL    ;asegurar HL
                                         salto retorno a HL
0001   D5              PUSH   DE         ;asegurar DE
0002   C5              PUSH   BC         ;asegurar BC
0003   F5              PUSH   AF         asegurar AF
```

```
0004   E5              PUSH  HL          ;asegurar direc.
                                          salto de retorno
0005   7E              LD    A,(HL)      ;1.Byte instrucción
0006   FE ED           CP    &ED         ;analiza prefijo
0008   28 08           JR    Z,SKPPFX
000A   FE DD           CP    &DD
000C   28 04           JR    Z,SKPPFX
000E   FE FD           CP    &FD
0010   20 01           JR    NZ,NOPFX
0012   23      SKPPFX  INC   HL          ;saltar prefijo
0013   23      NOPFX   INC   HL          ;HL indica el campo
                                          de dirección
0014   5E              LD    E,(HL)      ;toma Offset Low-Byte
0015   23              INC   HL
0016   56              LD    D,(HL)      ;y High-Byte
0017   2B              DEC   HL          ;HL indica inicio de
                                          la direc. del campo
0018   EB              EX    DE,HL       ;indicador en campo
                                          de dirección después
                                          de DE,Offset tras HL
0019   C1              POP   BC          ;dirección salto de
                                          retorno(=dirección
                                          base)después de BC
001A   C5              PUSH  BC          ;y de nuevo a la pila
001B   09              ADD   HL,BC       ;sumar Offset a la
                                          dirección base
001C   EB              EX    DE,HL       ;indicador dirección
                                          de campo después de
                                          HL,eff.Adr.tras de
001D   73              LD    (HL),E      ;Eff.direc. Low-Byte
001E   23              INC   HL
001F   72              LD    (HL),D      ;y High-Byte en campo
                                          de dirección
0020   E1              POP   HL          ;dirección salto de
                                          retorno tras HL
0021   2B              DEC   HL          ;HL indica instruc.
                                          RST
0022   36 00           LD    (HL),&00    ;activar instrucción
                                          RST mediante NOP
0024   23              INC   HL          ;HL contiene de nuevo
                                          direc. salto retorno
0025   F1              POP   AF          ;tomar de nuevo AF
```

```
0026   C1           POP   BC        ;tomar de nuevo BC
0027   D1           POP   DE        ;tomar de nuevo DE
0028   E3           EX    (SP),HL   ;tomar de nuevo HL,
                                    dirección de salto
                                    de retorno en pila
0029   C9           RET             ;salto de retorno
```

El hecho de que la dirección base de este listado de assembler se encuentre en &0000, significa a primera vista, y seguramente como Vd. habrá pensado, que esta dirección no puede ser nunca ejecutable. El espacio RAM de &0000 a &003F es absolutamente intocable para los programas de usuario, porque contiene los vectores del sistema operativo más importantes y más utilizados. (Una excepción de ello es únicamente el ya nombrado vector RST &30).
Pero este hecho no ha de considerarse como un adesventaja, porque se trata de direcciones relativas, lo que significa que no representan direcciones de memoria absolutas, sino que se trata únicamente de la distancia con una dirección base concreta.
La indicación de direcciones relativas es usual en el ensamblado reubicable; no tiene ningún tipo de valor introducir direcciones absolutas si no se ha determinado la dirección base.
Si esto le molesta, imagínese simplemente una A en lugar del primer 0 en cada dirección. Con ello no cambia la efectividad del listado, ni el sentido de la función en el programa, excepto que también se puede efectuar con la dirección base &A000. Por lo demás, nuestro pequeño programa auxiliar es suficiente, pues no contiene ninguna instrucción única, que se remita a una dirección absoluta, y que sea reubicable desde el principio. Esto significa que se puede operar sin problemas, en cualquier espacio de memoria asequible para el programa.
En el siguiente apartado podrá conocer algo más sobre la dirección base.

A pesar de la mínima extensión del programa, se cumplen las exigencias determinadas.

Una particularidad clara de este programa, que todavía no hemos nombrado es la siguiente:

Como actúa de forma muy parecida a una ampliación de instrucciones del procesador Z80, deberemos prestar atención en que no se cambie ningún registro. Sólo es posible una aplicación verdaderamente universal, cuando el subprograma no influye de ningún modo sobre el registro. Por esta razón, se salvarán al principio del programa, una serie de registros utilizados (PUSH), y se tomarán de nuevo de la pila, antes del salto de retorno (POP).
Aún debemos nombrar otra particularidad de este programa de reubicación, que no se considera en la mayoría de casos, pues si conduce a un error, se sigue buscando indefinidamente, si no se sabe lo siguiente: puesto que el sistema operativo del ordenador CPC debe desconectar la ROM inferior, para poder llegar al vector para la instrucción RST (que más adelante señala hacia nuestro programa), esta ROM se encontrará desconectada tras la llamada de RST &30, incluso en el caso de que anteriormente estuviese conectada.

23.5 El programa cargador

Antes de describir la función con un ejemplo, deberíamos aclarar un par de cuestiones importantes en relación con la forma de operar del programa: "¿Cómo llega el programa al procedimiento de ejecución requerido?"(El listado no contiene ninguna tarea para la dirección base del programa), y "¿Cómo se consigue que con la llamada del RST &30 se salte realmente al programa de reubicación?".

Primero responderemos la segunda cuestión. Si la instrucción RST &30 tiene sólo un byte y se ejecuta rápidamente, corresponde exactamente a la instrucción &CALL &0030. En la posición &0030 de la RAM se encuentran ocho bytes a disposición del usuario, y éste puede utilizar de &0030 hasta &0037 para la llamada de la rutina RST. Como una rutina puede ocupar apenas ocho bytes, se debe colocar un vector en la rutina deseada. Esta consta únicamente de una instrucción de salto directa (JP).

Se deben introducir sólo el opcode correspondiente a JP (&C3) en &0030, y la dirección de salto de nuestro programa en &0031 y &0032. Siempre que se ejecute un RST &30, se solicitará mediante esta instrucción de salto, la rutina correspondiente.
Como para activar el vector RST en nuestra rutina, necesitamos su dirección de inicio, (que es igual que la dirección base), volvemos a la primera cuestión: ¿Cómo se comunica la dirección base, y cómo reacciona el programa a partir de ese momento?

En la asignación de nuestro programa auxiliar en la memoria, nos encontramos con que es completamente reubicable sin necesidad de indicaciones especiales. Con ello es posible escribir un cargador para nuestro programa sin demasiado esfuerzo, que nos lleve automáticamente al lugar más favorable: al final superior de la memoria libre.
Para cargar el programa de un modo simple utilizaremos una variante del conocido cargador BASIC. Este cargador no comportará demasiada dificultad a causa de la corta extensión del programa auxiliar.

He aquí el programa cargador:

```
10 REM Cargador BASIC para programa auxiliar de relocalizaci
on Ver.1.0
20 REM (hr) 11/85
30 MEMORY HIMEM-42
40 FOR mptr=HIMEM+1 TO HIMEM+42
50 READ byte
60 POKE mptr,byte
70 cs=cs+byte
80 NEXT
90 IF cs<>&165C THEN PRINT CHR$(7);"Error en suma de prueba!
  ";HEX$(cs):STOP
100 REM activar vector RST:
105 IF HIMEM>32767 THEN temp=HIMEM-65536 ELSE temp=HIMEM
110 POKE &30,&C3:POKE &31,(temp+1) AND 255:POKE &32,(HIMEM+1
)/256
120 PRINT "Programa analisis de relocalizacion cargado corre
ctamente en ";HEX$(HIMEM,4)
130 END
1000 DATA &HE3,&HD5,&HC5,&HF5,&HE5,&H7E,&HFE,&HED
```

```
1005 DATA &H28,&H08,&HFE,&HDD,&H28,&H04,&HFE,&HFD
1010 DATA &H20,&H01,&H23,&H23,&H5E,&H23,&H56,&H2B
1015 DATA &HEB,&HC1,&HC5,&H09,&HEB,&H73,&H23,&H72
1020 DATA &HE1,&H2B,&H36,&H00,&H23,&HF1,&HC1,&HD1
1025 DATA &HE3,&HC9
```

Descripción del programa:

30: Mediante HIMEM se reserva espacio en la memoria.

40-80: El código máquina se lee a partir de las líneas DATA, y se almacena en la memoria reservada. En la línea 70 se efectúa la suma de pueba (cheksum).

90: Aquí se compara la suma de prueba introducida, con las anteriores, y en caso necesario, aparece un aviso de error (en el caso de una suma de prueba falsa), y se interrumpe el programa.

110: Asignación del vector RST &30 en la dirección de inicio del programa auxiliar.

120: Se comunica el resultado del proceso de carga con la condición de la dirección base del programa de reubicación.

13: Aquí finaliza el programa. En lugar de la instrucción END, se puede activar aquí la instrucción NEW, que borra automáticamente el programa que no es necesario después de la carga correspondiente. Esta versión del programa se debe almacenar naturalmente antes de la última inicialización, pues se borra con una nueva ejecución.

Este cargador BASIC nos proporciona todo lo necesario para la instalación de nuestro programa auxiliar:

la introducción de la dirección base (HIMEM menos extensión del programa), la reserva de espacio de memoria (instrucción MEMORY), la transmisión del programa al espacio reservado y el direccionamiento del vector. Pero aún lo necesitamos ...

23.6 Aplicación del programa de reubicación

¿Qué aspecto tendría un programa ensamblador, si lo conectamos con el programa auxiliar de reubicación dentro de su sistema operativo?
Los cambios, que se pueden realizar en un programa ensamblador normal, para poder adaptarlo a nuestro programa de reubicación, no son muy complicados. Sólo debemos incorporar los siguientes cambios:

* Cada instrucción, que se remite a una dirección dada, dentro de su campo de operandos, debe precederse directamente con la instrucción RST &30.

* En las mismas instrucciones se debe activar la dirección absoluta, mediante una dirección PC relativa de una extensión de 16 bits.
  Una dirección PC relativa no ofrece el estado de la dirección base del programa, sino la distancia en relación a la instrucción que se va a tomar (igual que la instrucción JR).

La inserción de la función RST en un programa dado, no presenta ningún problema, pero ¿qué pasa con las "direcciones relativas PC"?
También este método de direccionamiento es fácil de conseguir sin demasiada dificultad. La aplicación de direcciones relativas PC es muy ventajosa en nuestro caso, pues el programa recibe los datos completos que el programa principal necesita, mediante la pila, directa o indirectamente. Por esta razón, la transmisión de los programas a tratar, no es necesaria en algunas tablas de la dirección base.

Mediante la indicación de la distancia tenemos también a disposición un amplio espacio de memoria en todos los casos, que ocupa el espacio completo de la memoria del CPC (en el 6128, sólo un banco).
La entrada de direcciones relativas de PC en un programa ensamblador tampoco comporta dificultad. Unicamente debe introducir el signo $, y la dirección relativa del PC está lista.

Supongamos que tenga la siguiente llamada a subprograma, que se encuentra dentro del programa a tratar (la llamada remite a una dirección dentro del mismo programa):

```
        CALL   NUMOUT
```

Para preparar esta instrucción debemos escribir:

```
        RST    &30
        CALL   NUMOUT-$
```

El hecho de que utilicemos "-$", no significa que se trate de una instrucción mágica del assembler. El assembler ofrece la posibilidad de intervenir en el contador del programa, mediante el símbolo $, y así con la dirección de la instrucción, en cuya línea se encuentra $. La instrucción

```
        JP     $
```

por ejemplo, representa un bucle sin fin.

Vamos a substraer la instrucción de la dirección absoluta (junto con la dirección base del programa ofrecida por el assembler), en la dirección que deseamos intervenir, y así tendremos la dirección relativa.

Advertencia: La intervención en el contador de programa es posible en la mayoría de ensambladores. El signo dólar ($) representa en este caso una designación. También son posibles otras designaciones como *.

Si tratamos todas las instrucciones que dentro del programa se remiten al mismo fin de este modo, el programa está acabado, y se puede ejecutar en cualquier espacio de memoria libre, si se ha instalado el programa auxiliar de reubicación.

23.7 Un programa reubicable de ejemplo

Ahora queremos demostrarle la aplicación del programa de reubicación en un ejemplo práctico.
Lo que el programa lleva a cabo, no es nada especial: Después de la llamada con CALL, cuenta hacia atrás desde 100 a 0 y después vuelve al BASIC.

Aunque, partiendo del carácter ejemplar del programa, éste no es prácticamente activable, ofrece rutinas muy interesantes. El programa se ha realizado para demostrar la llamada de subprogramas, con excepción de la rutina del sistema operativo TXT OUTPUT totalmente independiente, que contiene una rutina de salida decimal propia: la rutina NUMOUT. Esta rutina de salida solicita continuamente la rutina DIV168, que ejecuta una división binaria. Más adelante trataremos más de cerca esta rutina, pues permite ejemplos claros para demostrar rutinas aritméticas complicadas.

Listado del programa de ejemplo:

```
; Programa de demostración de RELRST


0000   21 00 65   RRDEMO  LD    HL,101      ;valor inicio +1
0003   2B         LLOP00  DEC   HL          ;números
0004   E5                 PUSH  HL         ;seguro estado de
                                           contador
0005   F7                 RST   &30        Pr.relocalización
```

```
0006  CD 0D 00             CALL NUMOUT-$  ;ofrecer estado de
                                           contador
0009  E1                   POP  HL        ;toma otra vez estado
                                           de contador
000A  7C                   LD   A,H       ;HL
000B  B5                   OR   L         ;   =0?
000C  20 F5                JR   NZ,LOOP00 ;no:sigue contando
000E  3E 0D                LD   A,0DH     ;ofrecer CR
0010  C3 5A BB             JP   TXTOUT    ;TXT OUTPUT


          ;ofrecer número-16-Bit en HL en pantalla
          ;registro cambiado AF, HL, DE, BC, IX

0013  E5          NUMOUT   PUSH HL        ;asegura No.a ofrecer
0014  F7                   RST  &30
0015  21 4B 00             LD   HL,OUTBUF-$ ;borrar buffer de
                                           salida
0018  F7                   RST  &30
0019  11 48 00             LD   DE,OUTBUF+1-$
001C  01 00 05             LD   BC,5
001F  36 20                LD   (HL),' '
0021  ED B0                LDIR
0023  E1                   POP  HL        ;tomar de nuevo No.
                                           a ofrecer
0024  F7                   RST  &30
0025  DD 21 3F 00          LD   IX,OUTBUF+4-$ ;IX,indica buffer
                                           de salida
0029  0E 0A                LD   C,10      ;constante
002B  F7          LOOP01   RST  &30
002C  CD 1D 00             CALL DIV168-$  ;dividir HL por 10
002F  C6 30                ADD  A,'0'     ;resto división--->
                                           cifra ASCII
0031  DD 77 00             LD   (IX),A    ;escribir cifra en
                                           buffer
0034  DD 2B                DEC  IX
0036  EB                   EX   DE,HL     ;cociente es un No.
                                           nuevo
0037  7C                   LD   A,H       número
0038  B5                   OR   L         ;   =0?
0039  20 F0                JR   NZ,LOOP01 ;no:sigue transfor.
003B  F7                   RST  &30
```

```
003C  21 24 00           LD   HL,OUTBUF-$ ;ofrecer buffer
003F  06 05              LD   B,5         ;contador para carac.
0041  7E        LOOP02   LD   A,(HL)      ;tomar caracter
0042  CD 5A BB           CALL TXTOUT      ;ofrecer caracter
0045  23                 INC  HL          ;caracter siguiente
0046  10 F9              DJNZ LOOP02      ;sigue,si no está
                                          listo
0048  C9                 RET              ;final de salida


; División 16 Bits por 8 Bits
; Entrada:Dividiendo en HL,Divisor en C
; Salida:Cociente en DE,Resto en A
; Registro Cambiado: AF, HL, DE, BC

0049  11 00 00  DIV168   LD   DE,0        ;prepa. reg.-cociente
004C  06 10              LD   B,16        ;contador de Bit
004E  AF                 XOR  A           ;preparar reg. de
                                          trabajo y resto
004F  CB 23     LOOP03   SLA  E           ;cociente
0051  CB 23              RL   D           ;desplazamiento izq.
0053  CB 25              SLA  L           ;dividiendo
0055  CB 14              RL   H           desplazamiento izq.
0057  17                 RLA              ;Bit de mayor valor
                                          en acumulador
0058  B9                 CP   C           substracción posible?
0059  38 02              JR   C,SKIP00    ;no:salto
005B  13                 INC  DE          ;aumentar cociente
005C  91                 SUB  C           ;substraer divisor
005D  10 F0     SKIP00   DJNZ LOOP03      ;no está listo:sigue
005F  C9                 RET
0060            OUTBUF   DEFS 5           ;5Bytes buffer salida
```

En el listado del programa de demostración encontrará de nuevo las direcciones relativas ya conocidas.

En el programa de ejemplo, todas las instrucciones que contienen una dirección absoluta dentro del programa, se han precedido de la instrucción RST &30. Las direcciones se han transformado en direcciones PC relativas mediante la incorporación de "-$". Un ejemplo de ello son todas las instruciones, que se refieren a la dirección del buffer de salida (OUTBUF) (en las direcciones &0014, &0018, &0024 y &003C). En estas intervenciones se demuestra que también es posible la aplicación de un offset (p.e. en OUTBUF+4) sin problemas en el programa de reubicación. Las instrucciones de carga, que no contienen ninguna dirección relacionada con el programa, como p.e. la carga de la constante en la dirección &0000, no se han cambiado. La instrucción LD IX,OUTBUF+1-$ en la dirección &0024 demuestra cómo la ampliación RST se sitúa delante del prefijo. El programa auxiliar lo reconoce y lo interpreta, con la intervención en el campo de direcciones de la instrucción tratada. (Ver direcciones &0005 a &0013 en el listado del programa de reubicación.)
Las instrucciones CALL para los subprogramas NUMOUT y DIV168 se han precedido con un RST &30. Pero no se han tratado las llamadas de la rutina TXT OUTPUT, porque se refieren al sistema operativo, y no al programa.
Tampoco se han tratado las instrucciones de salto relativas JR y DJNZ. Esto no es necesario porque no se refieren a una dirección absoluta.

El dicho "mejor que sobre a que falte" no puede aplicarse al programa de reubicación. Las instrucciones RST &30 activadas innecesariamente destruyen el código que les sigue y provocan el bloqueo del programa. Pero si en un programa de reubicación se olvida de un RST &30, se producirá con toda seguridad el mismo resultado que en el caso anterior.

Vamos a finalizar la teoría. Con el siguiente cargador BASIC podrá analizar el programa de ejemplo en distintas direcciones de ejecución:

```
10 INPUT "Direccion de base";base
20 oldhimem=HIMEM
30 MEMORY basis-1
40 FOR mp=basis TO basis+100
50 READ byte
60 POKE mp,byte
70 cs=cs+byte
80 NEXT
90 IF cs<>&2751 THEN PRINT "Error en suma de prueba!":STOP
100 CALL basis
110 PRINT "Listo!"
120 MEMORY oldhimem
130 END
1000 DATA &21,&65,&00,&2B,&E5,&F7,&CD,&0D
1005 DATA &00,&E1,&7C,&B5,&20,&F5,&3E,&0D
1010 DATA &C3,&5A,&BB,&E5,&E7,&21,&4B,&00
1015 DATA &F7,&11,&48,&00,&01,&05,&00,&36
1020 DATA &20,&ED,&B0,&E1,&F7,&DD,&21,&3F
1025 DATA &00,&0E,&0A,&F7,&CD,&1D,&00,&C6
1030 DATA &30,&DD,&77,&00,&DD,&2B,&EB,&7C
1035 DATA &B5,&20,&F0,&F7,&21,&24,&00,&06
1040 DATA &05,&7E,&CD,&5A,&BB,&23,&10,&F9
1045 DATA &C9,&11,&00,&00,&06,&10,&AF,&CB
1050 DATA &23,&CB,&12,&CB,&25,&CB,&14,&17
1055 DATA &B9,&38,&02,&13,&91,&10,&F0,&C9
1060 DATA &00,&00,&00,&00,&00
```

Este cargador BASIC pregunta al principio por las direcciones base deseadas, reserva un espacio de memoria en ese lugar y finalmente copia el programa-ejemplo. Antes de activar de nuevo HIMEM mediante la instrucción MEMORY, se almacenará el valor anterior de HIMEM para conseguir el estado original al finalizar el programa máquina.

Para que este programa pueda ejecutarse, debe haberse instalado previamente el programa auxiliar de reubicación. De lo contrario, se produciría un bloqueo del ordenador.
Con este cargador BASIC puede funcionar en cualquier dirección que corresponda a la operación del programa. Las direcciones de menos extensión de los programas BASIC son principalmente de &1000 a &9FFF. En este caso también se debe prestar atención, en que no se solape el programa auxiliar de reubicación con la carga del programa de ejemplo. La dirección base superior permitida para el programa de ejemplo es la dirección base de programa auxiliar (que se ofrecerá mediante el cargador del programa auxiliar) con relación a la extensión del programa ejemplo (101 bytes). Teniendo en cuenta estos límites, el programa se puede ejecutar en cualquier dirección base. Si disminuimos la instrucción RST &30 del programa de ejemplo, éste puede transformarse en un programa máquina sujeto a una dirección completamente normal. Si comparamos el tiempo, constataremos que en lo relativo a la velocidad, no existe ninguna diferencia.

Después de este intenso tratamiento del programa de ejemplo, ya no debería representarle ningún problema la aplicación del programa auxiliar de reubicación.

23.7.1 Los subprogramas del programa ejemplo

Como ya hemos dicho anteriormente, ahora trataremos más de cerca los subprogramas NUMOUT y DIV168 del programa-ejemplo.
No podemos ofrecer una explicación completa de las rutinas aritméticas en este capítulo, pues sobrepasaríamos su marco. Por ello daremos sólo algunas indicaciones sobre la función de las rutinas, para que Vd. pueda aplicarlas para sus propias tareas.

De todas formas, esperamos que esta descripción junto con los comentarios sobre las rutinas en el listado le ayuden en la comprensión de estas rutinas.
En cualquier caso, se puede disponer de estas rutinas, especialmente la rutina de división, si no representan unas exigencias concretas. Los comentarios en el listado de rutinas informan sobre la entrega de los parámetros de entrada y salida, y sobre los parámetros cambiados. Las dos rutinas se llaman simplemente con la instrucción CALL.

La rutina de salida NUMOUT

Esta rutina transforma el número binario de 16 bits en el registro HL, a modo decimal. La salida se efectúa en un campo de cinco caracteres a partir de la posición actual del cursor; las posiciones de salida no utilizadas, se indican con un carácter vacío. El número se trata sin condicionamientos; por ello se pueden ofrecer valores enteros de 0 a 65535.

La rutina utiliza el buffer de salida largo OUTBUF para la salida de cinco caracteres (bytes). Esto no es sólo necesario para un buen direccionamiento, también lo es, porque el número de esta rutina se ha constituído hacia atrás. Las cifras se escriben primero en el buffer empezando por atrás y al final se ofrecen completas.

Al principio de la rutina de salida se rellena el buffer de salida con caracteres vacíos (direcciones &0013 hasta &0023). Esto se ha llevado a cabo en este caso por un efecto falso de la dirección LDIR. A través de la secuencia de instrucciones

```
LD      HL,START
LD      DE,START+1
LD      BC,EXTENSION-1
LD      (HL),BYTE
LDIR
```

se puede rellenar un bloque a partir de la dirección START de la longitud EXTENSION con el valor BYTE, donde el bloque casi llega a copiarse a sí mismo. Este es un método relativamente rápido (aunque no el más rápido), y de fácil aplicación, para rellenar una zona de memoria con una secuencia de bytes constante cualquiera, es decir, incluso con un único byte.

Después se inicializa el registro IX como indicador del final del buffer, y el registro C se carga con la constante necesaria para la transformación.

Finalmente se inicia la transformación a partir de la etiqueta LOOP01. Para ello se toma de nuevo el número dividido por 10 (DIV168) y el resto de la división resultante después de la transformación en una cifra ASCII (ADD A,'0') se almacena en el buffer (LD (IX),A); finalmente se activa el buffer en un carácter hacia delante (DEC IX). El cociente entero resultante de la rutina de división en el registro DE, se almacena al principio del bucle y antes del salto de retorno, como nuevo número almacenado en HL (EX DE,HL).
Este proceso se repite hasta que el cociente, después de la ejecución de un bucle, se encuentra en cero. El test nulo utilizado en este caso para el registro de 16 bits con las instrucciones LD y OR, es un método de test simple y rápido para el registro HL, DE y BC, que está compuesto especialmente en relación con las instruciones INC y DEC de 16 bits, ya que estas instrucciones no influyen los flags.

Si el cociente es nulo, no hay más cifras para introducir. La transformación ha terminado y el número aparece adosado a la derecha, precedido de caracteres vacíos, en el buffer. El contenido completo del buffer se editará en un bucle simple (LOOP02) mediante la rutina de sistema TXT OUTPUT en la pantalla.

Para aclarar mejor el sentido del trabajo de la rutina de salida, observe en el ejemplo de la transformación del número 523, el contenido de los registros decisivos y del buffer, mientras se lista la ejecución del bucle LOOP01 en la dirección &0036.

Los contenidos del registro se introducen como decimales, pues el contenido del buffer contiene los caracteres ASCII correspondientes. El contenido del registro HL no se introduce realmente en la dirección &0036, sino antes de la llamada de la rutina de división en la dirección &002B, ya que se cambia mediante la rutina de división.

Ejecución	HL	DE	A	Buffer
1	523	52	3	3
2	52	5	2	23
3	5	0	5	523

Después de la tercera ejecución del bucle, el cociente de DE es igual a cero, y con ello finaliza el bucle de transformación.

Al final de la rutina se produce el salto de retorno en el programa de llamada mediante RET.

La rutina de división DIV168

La rutina divide un número binario de 16 bits por uno de 8 bits y ofrece un cociente de 16 bits, y un resto de división de 8 bits. Todos los números son enteros.

Para entender el funcionamiento de esta rutina son necesarios unos buenos conocimientos sobre la aritmética binaria. Debemos renunciar a una explicación detallada sobre este tema pues sobrepasaría el marco de este capítulo.
Como indicación para el funcionamiento de esta rutina de división, diremos sólo que funciona según el conocido principio de la división binaria.
No confunda el sentido de esta rutina. Una división binaria es una de las rutinas elementales más complicadas del lenguaje máquina.

Pruebe alguna vez esta rutina con algunos ejemplos en papel, y podrá experimentar su funcionamiento. Yo personalmente, empecé a comprender la división binaria cuando me inicié en la programación.

23.8 Los límites de este método de reubicación

En este apartado resumiremos las limitaciones que se han ido presentando a lo largo de este capítulo.

Con nuestro pequeño programa de reubicación podemos efectuar cualquier tipo de intervención en un código objeto sin problemas, si su dirección se encuentra en el campo de operandos de una instrucción. La cosa se complica si deseamos intervenir en la dirección dada de un código objeto, dentro de una estructura de datos , p.e. una tabla de direcciones de salto. Aunque este problema también puede resolverse.

Para ello, el programa máquina debe saber en qué lugar de la memoria se encuentra. De este modo, se debe solicitar únicamente el subprograma:

```
GETPC    POP   HL
          JP   (HL)
```

Después de la solicitud con CAL GETPC, el subprograma toma su dirección de salto de retorno de la pila, mediante la instrucción POP, y salta a ella mediante JP (HL). Esto tiene el mismo efecto que una simple instrucción RET, pero devolviendo además a HL la dirección de salto de retorno, es decir, la dirección colocada directamente detrás de la instrucción CALL que llama al subprograma. Con ello ya disponemos prácticamente de todo lo necesario.
Con la secuencia de instrucciones

```
        RST     &30
        CÀLL    GETPC-$
BASE    RST     &30
        LD      (BSADR-$),HL
```

podemos averiguar la dirección realmente absoluta de la etiqueta BASE. Las entradas en una tabla de direcciones (p.e. una tabla de salto) pueden indicarse ahora relativamente en BASE. En lugar de

```
DW      dirección
```

deberemos escribir ahora

```
DW      dirección-BASE
```

en la tabla. Antes de acceder al objeto, se debe sumar la dirección de BASE, que se encuentra almacenada en la memoria de 16 bits BASADR, a la dirección relativa. Suponiendo que la dirección relativa se encontrara en el registro HL, podría efectuarse esto p.e. con la secuencia

```
RST     &30
LD      DE,(BASADR-$)
ADD     HL,DE
```

Los subprogramas y la memoria necesarios para ello, ocupan tan poco espacio, que apenas molestan.
En nuestro ejemplo ya hemos indicado el prefijo de reubicación (RST &30) siempre que fue necesario.

Con las indicaciones que hemos presentado en nuestro pequeño programa auxiliar de reubicación, se pueden llevar a cabo prácticamente todos los accesos referentes a direcciones dentro del mismo programa. Unicamente el acceso a la zona de direcciones de otro programa es todavía muy laborioso, porque no disponemos de la posibilidad de un acceso global.

23.9 La carga de un programa reubicable

Para cargar un programa pequeño tenemos a nuestra disposición . el cargador BASIC, como demostramos de forma práctica en nuestro programa-ejemplo.

El cargador BASIC se puede confeccionar sin dificultad, con ayuda de un programa. El cargador puede ser constituido directamente de tal forma, que permita averiguar la dirección BASE con ayuda de HIMEM, tal como sucede en el caso del cargador para el programa auxiliar de reubicación.

Para programas largos no es adecuado, pues la extensión del cargador BASIC aumenta rápidamente la extensión del programa (el cargador BASIC es casi tres veces más largo que el propio programa). Existe un programa cargador que está en situación de leer reubicablemente un fichero binario determinado en la dirección &0000, y donde también es razonable averiguar la dirección base mediante HIMEM. Este programa de servicio se encuentra aún en preparación, aunque es posible que se pueda editar en el diskette correspondiente a este libro.

COMMODORE



Ofrece un campo fascinante y amplio de problemáticas científicas. Para esto el libro contiene muchos listados interesantes: Análisis de Fornier y síntesis, análisis de redes, exactitud de cálculo, formateado de números, cálculo del valor PH, sistemas de ecuaciones diferenciales, modelo ladrón presa, cálculo de probabilidad, medición de tiempo, integración, etc.

64 en el campo de la Técnica y la Ciencia. 361 págs. P.V.P. 2.800,- ptas.



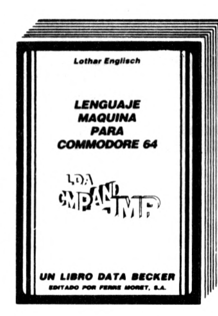
La obra Standard del floppy 1541, todo sobre la programación en disquettes desde los principiantes a los profesionales, además de las informaciones fundamentales para el DOS, los comandos de sistema y mensajes de error, hay varios capítulos para la administración práctica de ficheros con el FLOPPY, amplio y documentado Listado del Dos. Además un filón de los más diversos programas y rutinas auxiliares, que hacen del libro una lectura obligada para los usuarios del Floppy.

Todo sobre el Floppy 1541. Precio venta 3.200 ptas.



Un excelente libro, que le mostrará todas las posibilidades que le ofrece su grabadora de casettes. Describe detalladamente, y de forma comprensible, todo sobre el Datassette y la grabación en cassette. Con verdaderos programas fuera de serie: Autostart, Catálogo (¡busca y carga automáticamente!), backup de y a disco, SAVE de áreas de memoria, y lo más sorprendente: un nuevo sistema operativo de cassette con el 10-20 veces más rápido Fast Tape. Además otras indicaciones y programas de utilidad (ajuste de cabezales, altavoz de control).

El Manual del Cassette. 190 pág. P.V.P. 1.600,- ptas.



¡Por fin una introducción al código máquina fácilmente comprensible! Estructura y funcionamiento del procesador 6510, introducción y ejecución de programas en lenguaje máquina, manejo del ensamblador, y un simulador de paso a paso escrito en BASIC.

Lenguaje máquina para Commodore 64. 1984, 201 pág. P.V.P. 2.200,- ptas.



CONSEJOS Y TRUCOS, con más de 70.000 ejemplares vendidos en Alemania, es uno de los libros más vendidos de DATA BECKER. Es una colección muy interesante de ideas para la programación del Commodore 64, de POKEs y útiles rutinas e interesantes programas. Todos los programas en lenguaje máquina con programas cargadores en Basic.

64 Consejos y Trucos. 1984, 364 pág. P.V.P. 2.800,- ptas.



Este libro, contiene muchos interesantes programas de aprendizaje para solucionar problemas, descritos detalladamente y de manera fácilmente comprensible. Temas: progresiones geométricas, palanca mecánica, crecimiento exponencial, verbos irregulares, ecuaciones de segundo grado, movimientos de péndulo, formación de moléculas, aprendizaje de vocablos, cálculo de interés y su capitalización.

Manual escolar para su Commodore 64. 389 págs. P.V.P. 2.800,- ptas.



En el libro de los robots se muestran las asombrosas posibilidades que ofrece el CBM 64, para el control y la programación, presentadas con numerosas ilustraciones e intuitivos ejemplos. El punto principal: Cómo puede construirse uno mismo un robot sin grandes gastos. Además, un resumen del desarrollo histórico del robot y una amplia introducción a los fundamentos cibernéticos. Gobierno del motor, el modelo de simulación, interruptor de pantalla, el Port-Usuario cómodo del modelo de simulación, Sensor de infrarrojos, concepto básico de un robot, realimentación unidad cibernética, Brazo prensor, Oír y ver.

Robótica para su Commodore 64. 340 págs. P.V.P. 2.800 ptas.



Saberse apañar uno mismo, ahorra tiempo, molestias y dinero, precisamente problemas como el ajuste del floppy o reparaciones de la platina se pueden arreglar a menudo con medios sencillos. Instrucciones para eliminar la mayoría de perturbaciones, listas de piezas de recambio y una introducción a la mecánica y a la electrónica de la unidad de disco, hay también indicaciones exactas sobre herramientas y material de trabajo. Este libro hay que considerarlo en todos sus aspectos como efectivo y barato.

Mantenimiento y reparación del Floppy 1541. 325 págs. P.V.P. 2.800,- ptas.



Este es el libro que buscaba: un diccionario general de micros que contiene toda la terminología informática de la A a la Z y un diccionario técnico con traducciones de los términos ingleses de más importancia - los DICCIONARIOS DATA BECKER prácticamente son tres libros en uno. La increíble cantidad de información que contienen, no sólo los convierte en enciclopedias altamente competente, sino también en herramientas indispensables para el trabajo. El DICCIONARIO DATA BECKER se edita en versión especial para APPLE II, COMMODORE 64 e IBM PC.
El diccionario para su Commodore 64. 350 pág. P.V.P. 2.800,- ptas.



Casi todo lo que se puede hacer con el Commodore 64, está descrito detalladamente en este libro. Su lectura no es tan sólo tan apasionante como la de una novela, sino que contiene, además de listados de útiles programas, sobre todo muchas, muchas aplicaciones realizables en el C64. En parte hay listados de programas listos para ser tecleados, siempre que ha sido posible condensar «recetas» en una o dos páginas. Si hasta el momento no sabía que hacer con su Commodore 64, ¡después de leer este libro lo sabrá seguro!
El libro de ideas del Commodore 64. 1984, más de 200 páginas, P.V.P. 1.600,- ptas.



¿Ud. ha logrado iniciarse en código máquina? Entonces el «nuevo English» le enseñará cómo convertirse en un profesional. Naturalmente con muchos programas ejemplo, rutinas completas en código máquina e importantes consejos y trucos para la programación en lenguaje máquina y para el trabajo con el sistema operativo.
Lenguaje máquina para avanzados CBM 64. 1984, 206 pág. P.V.P. 2.200 ptas.



Este libro ofrece una amplia práctica introducción en el importante tema de la gestión de ficheros y bancos de datos, especialmente para los usuarios del Commodore 64. Con muchas interesantes rutinas y una confortable gestión de ficheros.
Todo sobre bases de datos y gestión de ficheros para Commodore-64. 221 págs. P.V.P. 2.200,- ptas.



Gráficos para el Commodore 64 es un libro para todos los que quieren hacer algo creativo con su ordenador. El contenido abarca desde los fundamentos de la programación de gráficos hasta el diseño asistido por ordenador (CAD).
Gráficos para el Commodore 64. 295 pags. P.V.P. 2.200,- ptas.



Para los usuarios que posean un VIC-20, C-64 o PC-128 este libro contiene gran cantidad de consejos, trucos, listados de programas, así como información sobre Hardware, tanto si usted dispone de una impresora de margarita o de matriz, como si tiene un Plotter VC-1520, el GRAN LIBRO DE IMPRESORAS constituye una inestimable fuente de información.
Todo sobre impresoras. 361 págs. P.V.P. 2.800,- ptas.



Con más de 60.000 ejemplares vendidos, ésta es la obra estándar para el COMMODORE 64. Todo sobre la tecnología, el sistema operativo y la programación avanzada del C-64. Con listado completo y exhaustivo de la ROM, circuitos originales documentados y muchos programas. ¡Conozca su C-64 a fondo!
64 Interno. 1984, 352 pág. P.V.P. 3.800,- ptas.



Con importantes comandos PEEK y POKE se pueden hacer también desde el Basic muchas cosas, para las que se necesitarían normalmente complejas rutinas en lenguaje máquina. Con una enorme cantidad de POKEs importantes y su posible aplicación. Para ello se explica perfectamente la estructura del Commodore 64: Sistema operativo, interpretador, página cero, apuntadores y stacks, generador de caracteres, registros de sprites, programación de interfaces, desactivación de interrupt. Además una introducción al lenguaje máquina. Muchos programas ejemplo.
PEEKS y POKES. 177 pág. P.V.P. 1.600,- ptas.



Este libro presenta una detallada e interesante introducción a la teoria, conceptos básicos y posibilidades de uso de la inteligencia artificial (1A). Desde un resumen histórico sobre las máquinas «pensantes» y «vivientes» hasta programas de aplicación para el Commodore 64.

Inteligencia artificial. 395 págs. 2.800,- ptas.



64, Consejos y Trucos vol. 2 contiene una gran profusión de programas, estímulos y muchas rutinas útiles. Un libro que constituye una ayuda imprescindible para todo aquél que quiera escribir programas propios con el COMMODORE.

Consejos y Trucos, Commodore 64. Vol. 2. 259 págs. 2.200,- ptas.



Este libro ofrece al programador interesado una introducción fácilmente comprensible para los tan extendidos Assembler PROFI-ASS, SM MAE y T.E-X.ASS. con consejos y trucos de gran utilidad, indicaciones y programas adicionales. Al mismo tiempo sirve de manual orientado a la práctica, con aclaraciones de conceptos importantes e instrucciones.

El Ensamblador. 250 páginas. 2.200,- ptas.



El libro de Primicias del Commodore 128 no ofrece solamente un resumen completo de todas las características y rendimientos del sucesor del C-64 y con ello una importante ayuda para su adquisición. Muestra, además, todas las posibilidades del nuevo equipo en función de sus tres modos de operación.

Todo sobre el nuevo Commodore 128. 250 págs. P.V.P. 2.200,- ptas.



El libro Commodore 128-Consejos y Trucos es un filón para cualquier poseedor del C-128 que desee sacar más partido a su ordenador. Este libro no sólo contiene gran cantidad de programas-ejemplo, sino que además explica de un modo sencillo y fácil la configuración del ordenador y de su programación.

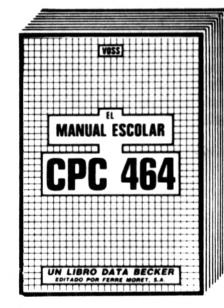
Commodore 128-Consejos y Trucos. 327 págs. 2.800,- ptas.

AMSTRAD



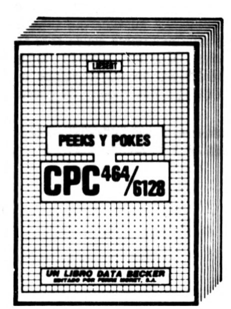
Ofrece una colección muy interesante de sugerencias, ideas y soluciones para la programación y utilización de su CPC-464: Desde la estructura del hardware, sistema de funcionamiento - Tokens Basic, dibujos con el joystick, aplicaciones de ventanas en pantalla y otros muchos interesantes programas como el procesamiento de datos, editor de sonidos, generador de caracteres, monitor de código máquina hasta listados de interesantes juegos.

CPC-464 Consejos y Trucos. 263 págs. P.V.P. 2.200,- ptas.



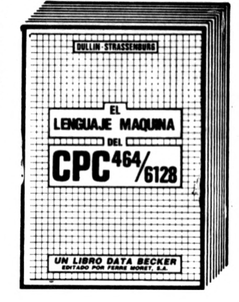
Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy compleja y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

CPC-464 El libro del colegio. 380 págs. P.V.P. 2.200,- ptas.



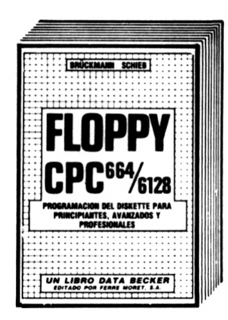
PEEKS, POKES y CALLS se utilizan para introducir al lector de una forma fácilmente accesible al sistema operativo y al lenguaje máquina del CPC. Proporciona además muchas e interesantes posibilidades de aplicación y programación de su CPC.

PEEKS y POKES del CPC 464/6128. 180 pág. P.V.P. 1.600,- ptas.



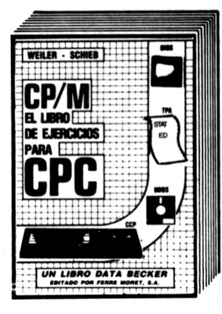
El libro del lenguaje máquina para el CPC 464/6128 está pensado para todos aquellos a quienes no les resulta suficiente con las posibilidades y rapidez del BASIC. Se explican aquí detalladamente las bases de la programación en lenguaje máquina, el funcionamiento del procesador Z-80 con sus respectivos comandos así como la utilización de las rutinas del sistema con abundantes ejemplos. El libro contiene programas completos de aplicación tales como Ensamblador. Desensamblador y Monitor, facilitando de esta manera la introducción del lector en el lenguaje máquina.

El Lenguaje Máquina del CPC 464/ 6128. 330 pág. P.V.P. 2.200,- ptas.



El LIBRO DEL FLOPPY del CPC lo explica todo sobre la programación con discos y la gestión relativa de ficheros mediante el floppy DDI-1 y la unidad de discos incorporada del CPC 664/6128. La presente obra, un auténtico stándart, representa una ayuda incomparable tanto para el que desee iniciarse en la programación con discos cómo para el más curtido programador de ensamblados. Especialmente interesante resulta el listado exhaustivamente comentado del DOS y los muchos programas de ejemplo, entre los que se incluye un completo paquete de gestión de ficheros.

El Libro del Floppy del CPC. 353 pág. P.V.P. 2.800,- ptas.



¡Dominar CP/M por fin! Desde explicaciones básicas para almacenar números, la protección contra la escritura, o ASCII, hasta la aplicación de programas auxiliares de CP/M, así como «CP/M interno» para avanzados, cada usuario del CPC rápidamente encontrará las ayudas e informaciones necesarias, para el trabajo con CP/M. Este libro tiene en cuenta las versiones CP/M 2.2, así como CP/M Plus (3.0), para el AMSTRAD CPC 464, CPC 664 y CPC 6128.

CP/M. El libro de ejercicios para CPC. 260 pág. P.V.P. 2.800,- ptas.

MSX



Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy completa y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

MSX el Manual Escolar. 389 págs. P.V.P. 2.800,- ptas.



El libro contiene una amplia colección de importantes programas que abarcan, desde un desensamblador hasta un programa de clasificaciones deportivas. Juegos superemocionantes y aplicaciones completas. Los programas muestran además importantes consejos y trucos para la programación. Estos programas funcionan en todos los ordenadores MSX, así como en el SPECTROVIDEO 318 328.

MSX Programas y Utilidades, 1985, 194 pág. P.V.P. 2.200,- ptas.



Las computadoras MSX no sólo ofrecen una relación precio/rendimiento sobresaliente, sino que también poseen unas cualidades gráficas y de sonido excepcionales. Este libro expone las posibilidades de los MSX de forma completa y fácil. El texto se completa con numerosos y útiles programas ejemplo.

MSX Gráficos y Sonidos, 250 págs. P.V.P. 2.800,- ptas.



Este libro contine una colección sin igual de trucos y consejos para todos los ordenadores con la nueva norma MSX. No sólo contiene las recetas completas, sino también los conocimientos básicos necesarios.

MSX - Consejos y Trucos. 288 págs. P.V.P. 2.200,- ptas.



El libro del Lenguaje Máquina para el MSX está creado para todos aquellos a quienes el BASIC se les ha quedado pequeño en cuanto a rendimiento y velocidad. Desde las bases para la programación en Lenguaje Máquina, pasando por el método de trabajo del Procesador Z-80 y una exacta descripción de sus órdenes, hasta la utilización de rutinas del sistema todo ello ha sido explicado en detalle e ilustrado con múltiples ejemplos en este libro.

El libro contiene, además, como programas de aplicación, un ensamblador un desensamblador y un monitor.

MSX Lenguaje Máquina. 306 págs. 2.200,- ptas.

ZX SPECTRUM



Una interesante colección de sugestivas ideas y soluciones para la programación y utilización de su ZX ESPECTRUM. Aparte de muchos peeks, pokes y USRs hay también capítulos completos para, entre otros, entrada de datos asegurado sin bloqueo de ordenador, posibilidades de conexión y utilización de microdrives y lápices ópticos, programas para la representación de diagramas de barra y de tarta, el modo de utilizar óptimamente ROM y RAM.

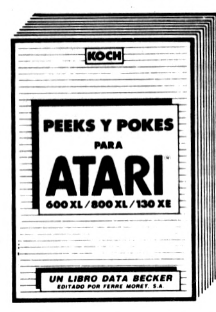
ZX Spectrum Consejos y Trucos, 211 pág. P.V.P. 2.200,- ptas.



Escrito para alumnos de los últimos cursos de EGB y de BUP, este libro contiene muchos programas para resolver problemas y de aprendizaje, descritos de una forma muy completa y fácil de comprender. Teorema de Pitágoras, progresiones geométricas, escritura cifrada, crecimiento exponencial, verbos irregulares, igualdades cuadráticas, movimiento pendular, estructura de moléculas, cálculo de interés y muchas cosas más.

ZX Spectrum el Manual Escolar. 389 págs. P.V.P. 2.200,- ptas.

ATARI



Tan interesante como el tema, es el libro que explica de forma fácilmente comprensible el manejo de Peeks y Pokes importantes, y representa un gran número de Pokes con sus posibilidades de aplicación, incluyendo además programas ejemplo. Al lado de temas como lo son la memoria de la pantalla, los bits y los bytes, el mapa de la memoria, la tabla de modos gráficos o el sonido, también se detalla de forma magnífica la estructura del ATARI 600XL/800XL/130XE.

Peeks y Pokes para ATARI 600XL/800XL/130XE. 251 pág. P.V.P. 2.200, ptas.



Una lograda introducción al sugestivo tema de los «juegos estratégicos». Desde juegos sencillos con estrategia fija a juegos complejos con procedimientos de búsqueda hasta programas con capacidad de aprendizaje —muchos ejemplos interesantes, escritos por supuesto de forma fácilmente comprensible. Con programas de juegos ampliamente detallados: NIM con un montón, bloqueo, hexapawn, mini-damas y muchos más.

Juegos estratégicos y cómo programarlos en el ATARI 600XL/800XL/130XE. 181 pág. P.V.P. 1.600,- ptas.



Jugar a aventuras con éxito y programarlas uno mismo - todo lo verdaderamente importante sobre el tema, lo contiene este guía fascinante que te lleva a través del mundo fantástico de las aventuras. El libro abarca todo el espectro, hasta las más sofisticadas aventuras gráficas llenas de trucos, acompañándolas siempre de numerosos programas ejemplo. Sin embargo la clave —al margen de muchas aventuras para teclear— es un generador de aventuras completo, mediante el cual la programación de aventuras se convierte en un juego de niños.

Aventuras - y cómo programarlas en el ATARI 600XL/800XL/130XE. 284 pág. P.V.P. 2.200,- ptas.



Muchos programas interesantes de soluciones de problemas y de aprendizaje, descritos de forma amplia y comprensible, y adecuados sobre todo para escolares. ¡Aquí el aprendizaje intensivo se convierte de una tarea divertida! Al margen de temas como los verbos irregulares, o las ecuaciones de segundo grado, un resumen corto de las bases del tratamiento electrónico de datos, y una introducción a los principios del análisis de problemas, completan este libro que debería obrar en posesión de cualquier escolar.

El libro escolar para ATARI 600XL/800XL/130XE. 389 pág. P.V.P. 2.800,- ptas.

OTROS TITULOS



El primer libro recomendado para escuelas de enseñanza de informática y para aquellas personas que quieren aprender la programación. Cubre las especificaciones del Ministerio de Educación y Ciencia para Estudios de Informática. Es el primer libro que introduce a la lógica del ordenador. Es un elemento de base que sirve como introducción para la programación en cualquier otro lenguaje. No se requieren conocimientos de programación ni siquiera de informàtica. Abarca desde los métodos de programación clásicos a los más modernos.

Metodología de la Programación. 250 págs. P.V.P. 2.200,- ptas.



La técnica y programación del Procesador Z80 son los temas de este libro. Es un libro de estudio y de consulta imprescindible para todos aquellos que poseen un Commodore 128, CPC, MSX u otros ordenadores que trabajan con el Procesador Z80 y desean programar en lenguaje máquina.

El Procesador Z80. 560 pág. P.V.P. 3.800,- ptas.



El tema de este libro es la técnica y programación de los procesadores de la familia 68000. Es una obra de consulta indispensable, un manual para todo programador que quiera utilizar las ventajas del 68000.

Técnica y programación para el procesador 68000. 516 págs. P.V.P. 3.800,- ptas.

RESPUESTA COMERCIAL

F.D. Autorización 6975
(B.O. de Correos N.° 80 de 26-7-85)

HOJA PEDIDO DE LIBRERIA

NO NECESITA SELLOS

A franquear en destino

FERRE MORET, S.A.

Apartado N.° 551. F.D.
08080 BARCELONA

RESPUESTA COMERCIAL

F.D. Autorización 6975
(B.O. de Correos N.° 80 de 26-7-85)

HOJA PEDIDO DE LIBRERIA

NO NECESITA SELLOS

A franquear en destino

FERRE MORET, S.A.

Apartado N.° 551. F.D.
08080 BARCELONA

Puesta al día de datos

EDITORIAL FERRER MORET, S.A. mantiene vivo y amplía el contenido informativo de sus libros y programas, mediante el envío de un servicio de puesta al día, junto con una síntesis noticiosa de la actualidad y perspectivas de la realidad informática española.
Agradecemos cualquier sugerencia o crítica que desee formular y que nos ayude a mejorar las ediciones. Muchas gracias.

¿Qué añadiría?
....

....

¿Qué suprimiría?
....

....

Observaciones
....

....

Título del libro
....

Nombre
....

Dirección Tfno.
....

Código Postal y Población Provincia
....

UN SERVICIO GRATUITO

Información y Pedidos

FERRE MORET, S.A. cuenta con un amplio fondo de libros y Software y mantiene un servicio de información por correo sobre las novedades que edita.
Agradecemos nos indique los temas que representan para Vd. mayor interes.

Libros	☐ ATARI	☐ MSX	☐ AMSTRAD
	☐ COMMODORE	☐ LENGUAJES	☐ APPLE
	☐ SINCLAIR	☐ IBM	☐ SOFTWARE

Si está interesado en recibir alguno de estos servicios, rellene y envíe la tarjeta correspondiente; no necesita franqueo. Muchas gracias.

DESEO RECIBIR EL LIBRO ....

EL PROGRAMA ....

☐ Adjunto cheque ☐ Contra reembolso
....

Nombre
....

Dirección Tfno.
....

Código Postal y Población Provincia
....

UN SERVICIO GRATUITO

EL CONTENIDO:

El segundo tomo del CPC Consejos y trucos es interesante para todos aquellos que tengan un 464, 664 o 6128, y que deseen obtener una serie de consejos para el BASIC, ampliación de instrucciones, y lenguaje máquina. Ofrecemos muchas rutinas auxiliares efectivas.

Extracto del contenido:

- Procedimiento de clasificación
- Gráficos tridimensionales
- Generador de menú
- Generador de máscaras de entrada
- Protección de programas propios
- Volcado de variables
- Hardcopy de gráficos
- Líneas BASIC creadas desde el BASIC
- Puntos importantes para la programación en lenguaje máquina
- Soft-Scrolling
- Interface de BASIC para los registros del Z80
- Rutinas del interpretador y del sistema operativo
- Compatibilidad entre los tres ordenadores CPC
- Programas reubicables en lenguaje máquina

ESTE LIBRO HA SIDO ESCRITO POR:

Holger Dullin y Hardy Strassenburg, equipo técnico que ha realizado trabajos interesantes para DATA-BECKER (El lenguaje máquina del CPC), así como Jochen Schneider, experto en programación y en comercio del proceso de datos de la casa DATA-BECKER, y Helmut Retzlaff, con amplia experiencia en el campo de los ordenadores Z80.

ISBN 84-86437-56-3

Dullin · Retzlaff · Schneider · Strassenburg /
CPC Consejos y trucos - 2

AMSTRAD


AMSTRAD
CPC
OCR
MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA
ACME
https://acpc.me/